Colección Manuales Docentes

Estructura de computadores. Prácticas de laboratorio

Arduino Zero para la construcción de un robot móvil como plataforma de prácticas

M.ª José Santofimia Romero
Juan Carlos López López
Xavier del Toro García (coords.)

Ediciones de la Universidad
de Castilla-La Mancha

ESTRUCTURA DE COMPUTADORES. PRÁCTICAS DE LABORATORIO

ARDUINO ZERO PARA LA CONSTRUCCIÓN DE UN ROBOT MÓVIL COMO PLATAFORMA DE PRÁCTICAS

ESTRUCTURA DE COMPUTADORES. PRÁCTICAS DE LABORATORIO

ARDUINO ZERO PARA LA CONSTRUCCIÓN DE UN ROBOT MÓVIL COMO PLATAFORMA DE PRÁCTICAS

M.ª José Santofimia Romero

Juan Carlos López López

Xavier del Toro García

(coords.)

Ediciones de la Universidad de Castilla-La Mancha

Cuenca, 2020

Edita: Ediciones de la Universidad de Castilla-La Mancha.

Colección MANUALES DOCENTES n.º 16

Ilustración de cubierta y composición: Jaime López Molina

Esta editorial es miembro de la UNE, lo que garantiza la difusión y comercialización de sus publicaciones a nivel nacional e internacional

I.S.B.N.: 978-84-9044-411-5

D.O.I.: http://doi.org/10.18239/manuales_2020.16.00

Hecho en España (U.E.) – *Made in Spain (U.E.)*

RESUMEN

El proyecto Arduino surge en el año 2005, como plataforma educativa y para la creación de proyectos interactivos, con el propósito de ofrecer una placa electrónica de prototipado de bajo coste y *open source*. El éxito de este proyecto hace que el uso de esta plataforma no se limite únicamente a entornos educativos y se haya extendido al prototipado de soluciones de automatización y control a pequeña escala. Su bajo coste, facilidad de aprendizaje y la amplia comunidad de usuarios, son probablemente las claves más importantes de su éxito.

En el ámbito universitario es cada vez más común encontrar prácticas de laboratorio relacionadas con la electrónica o la programación que utilizan Arduino como plataforma de experimentación. En nuestro caso, nuestra motivación para elegirla como plataforma de prácticas es que nos ofrece un computador lo suficientemente complejo a la par que sencillo para que, mediante la exploración y la práctica, los estudiantes de la asignatura Estructura de Computadores puedan conocer y familiarizarse con los distintos elementos que componen un computador.

Este manual de prácticas describe el proceso de desarrollo y programación de un robot móvil empleando Arduino Zero. A lo largo de este proceso el alumno podrá explorar, de un modo práctico, los aspectos más esenciales de la estructura de un computador (el sistema de memoria, entrada/salida, ABI, etc.). Los primeros capítulos se dedicarán a ofrecer la información necesaria para abordar los diferentes ejercicios que se propondrán en las sesiones de prácticas, de tal forma que para cada capítulo se dedicarán dos sesiones. En la primera sesión

se realizarán ejercicios guiados relacionados con el tema estudiado y durante la segunda sesión se realizará un ejercicio de evaluación similar a los de autoevaluación (con soluciones) propuestos en este manual. Finalmente, los anexos ofrecen información guiada para la instalación y configuración del entorno de desarrollo en sistemas GNU/Linux.

AGRADECIMIENTOS

Los autores quieren agradecer el apoyo económico de la Escuela Superior de Informática de Ciudad Real para la adquisición de los diferentes componentes de la plataforma de prácticas.

ÍNDICE

1. INTRODUCCIÓN 15

2. ELECCIÓN DE LA PLATAFORMA 17
   2.1. ¿Microcontrolador o microcomputador? 18
   2.2. Arduino a diferencia de otros microcontroladores 19
   2.3. ¿Cómo funciona Arduino? 19

3. CONSTRUCCIÓN DEL ROBOT MÓVIL 21

4. INTRODUCCIÓN A ESTRUCTURA DE COMPUTADORES 27
   4.1. La plataforma Arduino Zero 28
      4.1.1. El microcontrolador de Atmel 28
      4.1.2. El procesador y la arquitectura ARM 29
      4.1.3. El depurador EDBG 30
      4.1.4. Alimentación de la placa 31
      4.1.5. El sistema de memoria 32
      4.1.6. Entrada y Salida 33
      4.1.7. La programación 34
      4.1.8. El proceso de arranque 35
   4.2. Tipos de arquitectura: von Neumann y Harvard 35
      4.2.1. ¿Qué es un computador? 35
      4.2.2. Arquitectura de un computador 35
   4.3. Arduino Zero como ejemplo de arquitectura Harvard 38

5. MEMORIA 39
5.1. Gestión de la memoria en Arduino Zero 39
5.2. El Application Binary Interface (ABI) 42
5.2.1. El endianness 43
5.2.2. Reglas de alineamiento 44
5.2.3. La llamada a procedimientos 47
5.3. La pila 51

6. LENGUAJE MÁQUINA 53
6.1. El conjunto de instrucciones del Cortex M0+ 54
6.2. La sintaxis del lenguaje ensamblador 55
6.2.1. La lista de instrucciones 56
6.2.2. Operaciones lógicas 59

7. ENTRADA Y SALIDA 61
7.1. Técnicas de entrada/salida 62

8. SESIONES DE PRÁCTICAS 65

9. SESIÓN 1. EXPERIMENTANDO CON LA ARQUITECTURA HARVARD DE ARDUINO ZERO 67
9.1. "Hola, Mundo" 67
9.2. Explorando la memoria 69

10. SESIÓN 2. EVALUACIÓN DE LA ARQUITECTURA HARVARD 77
10.1. Sesión 2: Preguntas 77
10.2. Sesión 2: Respuestas 78
10.2.1. Pregunta 1 78
10.2.2. Pregunta 2 80
10.2.3. Pregunta 3 80
10.2.4. Pregunta 4 83
10.2.5. Pregunta 5 85

11. SESIÓN 3: MEMORIA 87
11.1. La pila 87
11.2. Alineamiento de datos en la pila 91

12. SESIÓN 4: EVALUACIÓN 93
12.1. Sesión 4: Preguntas 93
12.2. Sesión 4: Respuestas 93
12.2.1. Pregunta 1 93
12.2.2. Pregunta 2 94
12.2.3. Pregunta 3 95
12.2.4. Pregunta 4 96
12.2.5. Pregunta 5 97

13. SESIÓN 5: LENGUAJE MÁQUINA 99
13.1. Control del programa 99
13.2. Introduciendo retrasos empotrando ensamblador 100
13.3. Modos de direccionamiento 102

14. SESIÓN 6: EVALUACIÓN 105
14.1. Sesión 6: Preguntas 105
14.2. Sesión 6: Respuestas 106
14.2.1. Pregunta 1 106
14.2.2. Pregunta 2 107
14.2.3. Pregunta 3 107
14.2.4. Pregunta 4 108
14.2.5. Pregunta 5 108

15. SESIÓN 7: ENTRADA/SALIDA 111
15.1. Entrada: Sensores 111
15.1.1. Sensor de ultrasonidos 112
15.1.2. Sensor de infrarrojo 113
15.1.3. Sensor de luz 114

15.2. Salida: Actuadores 115
15.2.1. Zumbador 115
15.2.2. Servomotor 117
15.2.3. Miniservo 118
15.3. Entrada/Salida por interrupción 120

16. SESIÓN 8: EVALUACIÓN 123
16.1. Lectura del sensor infrarrojo 123
16.2. Control de los servos 124
16.3. Siguelíneas. 126

ANEXOS

ANEXO A: ARDUINO Y GNU/LINUX 131
Elementos necesarios. 131
Software de Arduino (IDE) 131
Configuración 132
Nuestro primer programa 133

ANEXO B: DEPURACIÓN DE PROGRAMAS CON EDBG . . 135
Ajustes previos en el IDE de Arduino. 136
OpenOCD y GDB 139
KDbg: Frontend para GDB 143

BIBLIOGRAFÍA 147

1
INTRODUCCIÓN

La robótica educativa es una disciplina pedagógica que ha ganado fuerza en los últimos años. En el proceso de diseño y desarrollo de robots los estudiantes desarrollan una serie de habilidades que extienden las propias asociadas a las ciencias y la tecnología. El desarrollo del pensamiento lógico, computacional, sistemático y estructurado capacita al estudiante para abordar, apropiadamente, la tarea de resolución de problemas.

Sin embargo, parece que el ámbito de aplicación de la robótica educativa se ha centrado en los ámbitos de la educación primaria y secundaria, teniendo un menor preso en el caso de la educación universitaria o superior. Aunque el estudiante universitario ya ha desarrollado a lo largo de trayectoria como estudiante muchas de las habilidades que la robótica educativa impulsa, ésta aún puede ayudar a potenciar muchas otras competencias más específicas de un pensamiento ingenieril.

Parece lógico pensar que el contexto universitario también pueda nutrirse de estas bondades, especialmente en estudios de ingeniería, donde el estudiante debe desarrollar habilidades que le permitan enfrentarse a tareas como la resolución de problemas, el razonamiento analítico y deductivo, la lógica o el trabajo colaborativo. A estas circunstancias se une el descontento, cada vez más generalizado por parte de los estudiantes hacia el uso de herramientas de simulación como herramienta de laboratorio. Los simuladores son una herramienta muy útil de apoyo a las experiencias prácticas, pero no deben ser la única herramienta de trabajo. Los entornos simulados suelen utilizarse porque la realidad es demasiado compleja

y el estudiante se perdería en la complejidad de detalles no relevantes respecto al problema que se desea estudiar. Por ejemplo, en el caso de los procesadores, el uso de procesadores reales lleva aparejado una serie de cuestiones (ajenas a los conceptos fundamentales de la estructura de un computador) que haría inviable su uso como herramienta de laboratorio. Sin embargo, los entornos de simulación presentan a veces escenarios demasiado simplificados que abstraen al estudiante de la realidad abordada. Por lo tanto, el uso de simuladores puede considerarse como una herramienta de apoyo, pero en la medida de lo posible, es deseable trabajar con elementos reales.

Sobre la base de los avances conseguidos por el uso de la robótica educativa en las aulas de primaria y secundaria, este laboratorio pretende adaptar dicho enfoque a un entorno de educación superior y, más concretamente, al laboratorio de la asignatura de Estructura de Computadores impartida en el primer curso de los estudios de Grado en Ingeniería Informática.

La plataforma elegida, en este caso, ha sido Arduino Zero que cuenta con un procesador ARM de 32 bits y un chip de depuración integrado como características más reseñables. Además, se utilizarán una serie de elementos accesorios para conseguir desarrollar un robot que se pueda emplear en diversos tipos de aplicaciones, como por ejemplo un sigue líneas.

2
ELECCIÓN DE LA PLATAFORMA

Desde el curso 2007/2008 el laboratorio de Estructura de Computadores venía impartiéndose utilizando la videoconsola Nintendo DS. Pese al acierto que supuso en su día el uso de esa plataforma, el paso del tiempo ha hecho que la misma quede obsoleta y la falta de soporte de la comunidad *homebrew* haga cada vez más complicado el proceso de configuración del entorno de desarrollo.

Durante los últimos cursos, además, se había introducido el uso de librerías que simplificaban enormemente el desarrollo de gráficos, dando como resultado juegos mucho más vistosos, pero abstrayendo, por otro lado, al alumno de una de las cuestiones más interesantes de la Nintendo DS como plataforma, como era la gestión de la memoria de gráficos.

El cambio del plan de estudios también tuvo un impacto importante en la parte de laboratorio, concebido en el antiguo plan para el doble de esfuerzo. El laboratorio tuvo que adaptarse a esa reducción y, desde el curso 2015/2016, a un nuevo aumento de carga en la parte de laboratorio.

Todas estas cuestiones han motivado la búsqueda de una nueva plataforma sobre la que poner en práctica los conceptos estudiados en teoría, pudiendo a la vez disfrutar de la experiencia de trabajar con dispositivos reales y no meros simuladores.

El proceso de elección de la plataforma no ha sido sencillo y del análisis del mismo pueden extraerse importantes conclusiones que consideramos interesantes para el estudiante y por este motivo las presentamos aquí.

2.1. ¿MICROCONTROLADOR O MICROCOMPUTADOR?

Esta fue la primera pregunta a la que debimos dar respuesta ¿Qué es más apropiado, un microcontrolador o un microcomputador? O, traducido a las plataformas más conocidas en estos ámbitos: Arduino o Raspberry Pi.

Por su precio y popularidad en la red, ambas parecían perfectas candidatas *a priori*, por lo que había que realizar un análisis más profundo para determinar la idoneidad de una sobre otra para el contexto del laboratorio de Estructura de Computadores.

Una de las primeras cuestiones en contra del uso de Raspberry Pi es que ésta, al ser un microcomputador, presenta una mayor complejidad cuando la comparamos con Arduino que es un microcontrolador. Esto supone que Raspberry Pi, como casi cualquier otro equipo de propósito general implementa una arquitectura tipo von Neumann mientras que Arduino implementa una arquitectura Harvard. Arduino, como cualquier otro microcontrolador, está pensado para ser programado para un único propósito, mientras que Raspberry Pi dispone de un sistema operativo sobre el que se ejecutarán diferentes tareas paralelamente.

Por su parte, tanto Raspberry Pi como algunos modelos de Arduino utilizan un procesador ARM, con las ventajas que esto ofrece desde el punto de vista didáctico: un ABI bien documentado, un conjunto de instrucciones sencillo, popularidad ya que la mayor parte de los dispositivos empotrados móviles utilizan tecnologías ARM, etc.

Por lo tanto, parece que la simplicidad de una plataforma como Arduino se impone a la potencia y complejidad de Raspberry Pi. Sin embargo, el proceso de elección no termina aquí porque el abanico de placas Arduino es cada vez mayor y había que elegir una de ellas.

Dentro de los diferentes procesadores que Arduino implementa, los ARM nos permiten trabajar con una arquitectura de 32 bits. Más concretamente, Arduino Zero cuenta, además, con un sistema de memoria que supera en capacidad, especialmente en la memoria tipo SRAM, a lo ofrecido por otras versiones de Arduino. Pero si hay un motivo por el que nos hemos decantado especialmente por esta plataforma es porque ésta incorpora un chip de depuración que evita tener que adquirir y montar un hardware adicional para realizar este tipo de tareas. Aunque la depuración está principalmente orientada a la búsqueda y solución de errores en la codificación de un programa, en nuestro caso, la depuración se utilizará como mecanismo para explorar la arquitectura (especialmente el sistema de memoria). El chip EDBG viene integrado en la placa Arduino Zero y da soporte a estas tareas. Además, aunque de manera accesoria, hemos valorado

especialmente el bajo consumo de la placa que lo hace ideal en aplicaciones en el ámbito del *Internet of Things* (IoT), pudiendo éste ser utilizado para futuros proyectos de aquellos estudiantes que decidan adquirir la placa.

2.2. ARDUINO A DIFERENCIA DE OTROS MICROCONTROLADORES

Aunque en la sección anterior hemos simplificado la discusión a Arduino vs. Raspberry Pi, la lista de microcontroladores no se limita a placas Arduino. Sin embargo, al margen de ser una plataforma pensada especialmente para la comunidad educativa, ha habido otros motivos por los que nos hemos decantado por Arduino, frente a otros microcontroladores tal y como justificamos a continuación:

- Es una plataforma sencilla y accesible, lo que hace que sea la plataforma elegida para miles de proyectos y aplicaciones que pueden servirnos de inspiración y guía.
- Sigue la filosofía *open source* (código abierto) de tal forma que tanto el ensamblaje de la placa como el código fuente son de acceso público.
- Por otro lado, nos encontramos con la comunidad que se ha formado a su alrededor. La comunidad Arduino se desarrolla y enriquece a partir del trabajo de sus colaboradores, siendo todo este conocimiento público y accesible a cualquier persona.
- El software Arduino (IDE) es multiplataforma, corre bajo los sistemas operativos GNU/Linux, Windows y Macintosh OSX, cuando la mayoría de los microcontroladores están limitados al sistema operativo Windows.
- Otro punto a su favor es la sencillez y accesibilidad del lenguaje de programación que hace que sea de fácil uso para principiantes y lo suficientemente flexible para usuarios avanzados.
- Arduino es un hardware de bajo coste, por lo que es factible que aquellos estudiantes interesados en adquirir su propia plataforma no tengan que afrontar un importante desembolso. Además, la plataforma adquirida podrá reutilizarse para proyectos propios, además de para este laboratorio.

2.3. ¿CÓMO FUNCIONA ARDUINO?

Arduino es una placa de circuito impreso que dispone de todos los elementos necesarios para poderse conectar diferentes periféricos (sensores, actuadores, comunicaciones, etc.). Así, su microcontrolador posee entradas y salidas digitales, analógicas y para ciertos protocolos de comunicación (I^2C, SPI, etc.).

Arduino utiliza un conversor de serie a USB, por lo que a la hora de conectarse a un ordenador simplemente se necesita un simple conector USB. El ordenador, sin embargo, verá nuestro Arduino como un dispositivo conectado al puerto serie.

El microcontrolador hace las funciones de "*cerebro*" de la placa Arduino. La programación del microcontrolador se realiza a través del IDE de Arduino. Una vez escrito el programa y compilado, éste se carga utilizando la conexión USB entre el ordenador y Arduino. El lenguaje Arduino, basado en *Wiring* y derivado de C, es mucho más amigable que el lenguaje ensamblador, utilizado por otros microcontroladores. El intercambio de información entre el usuario y la placa de Arduino se puede realizar utilizando las herramientas que el IDE trae incorporadas, como la comunicación serie.

Existen cientos de modelos de sensores y módulos electrónicos que se pueden conectar a Arduino, especialmente los conocidos como *shields*, que son placas con el mismo factor de forma que pueden ser directamente conectadas sin necesidad de cableado.

Los modelos más potentes de Arduino, como el Arduino Yun, incorporan un microprocesador que permite el uso de sistemas operativos como Linux, ampliándose así el poder de procesamiento y las capacidades en varios órdenes de magnitud.

3
CONSTRUCCIÓN DEL ROBOT MÓVIL

En este capítulo se describe el proceso de construcción del robot móvil que se empleará como plataforma de prácticas, enumerando los distintos elementos que lo compondrán y su ensamblaje.

Juntamente con la placa basada en microcontrolador, que es el elemento principal para la asignatura, el robot constará de una estructura mecánica, un conjunto de sensores y actuadores, la electrónica adicional necesaria y el sistema de alimentación.

Existen en el mercado una gran variedad de productos que, con un coste asequible, permiten la construcción de robots móviles. En este caso se ha decidido como estructura de base el robot Printbot Evolution de la empresa BQ[1] (ver Figura 1 y Figura 2). Se trata de un diseño libre, cuyas partes pueden fabricarse mediante impresora 3D. Además, incorpora un buen número de sensores y actuadores con un coste total razonable y una calidad en cuanto a componentes y materiales adecuada.

La estructura mecánica del Printbot Evolution consiste en:

- Un chasis de metacrilato sobre el que se montan el resto de los componentes.
- Dos ruedas con servomotores independientes para controlar la dirección.
- Un porta pilas cuya carcasa sirve como tercer punto de apoyo del robot.
- Un soporte para montar un sensor de ultrasonidos sobre un miniservo.

1 https://www.bq.com/es/printbot-evolution

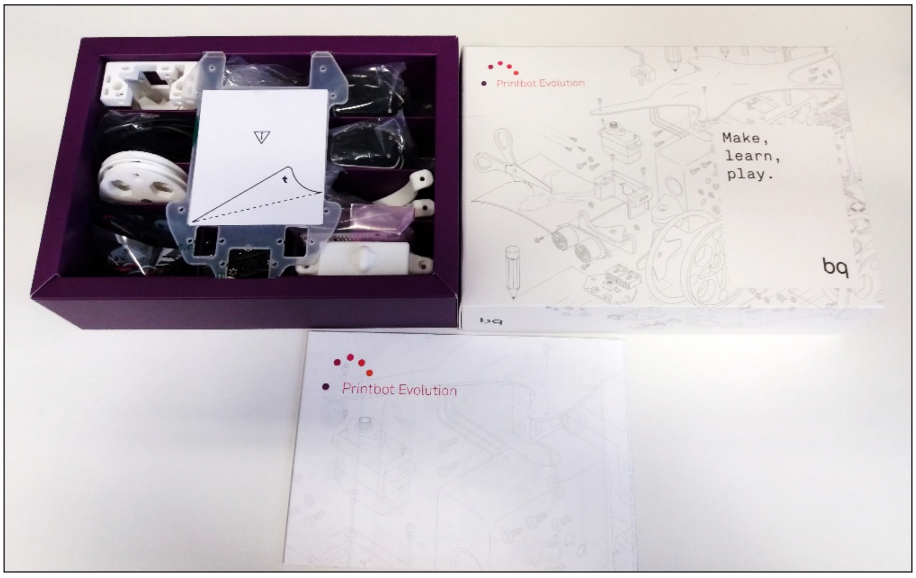

Figura 1. Caja con el contenido del robot Printbot Evolution de BQ

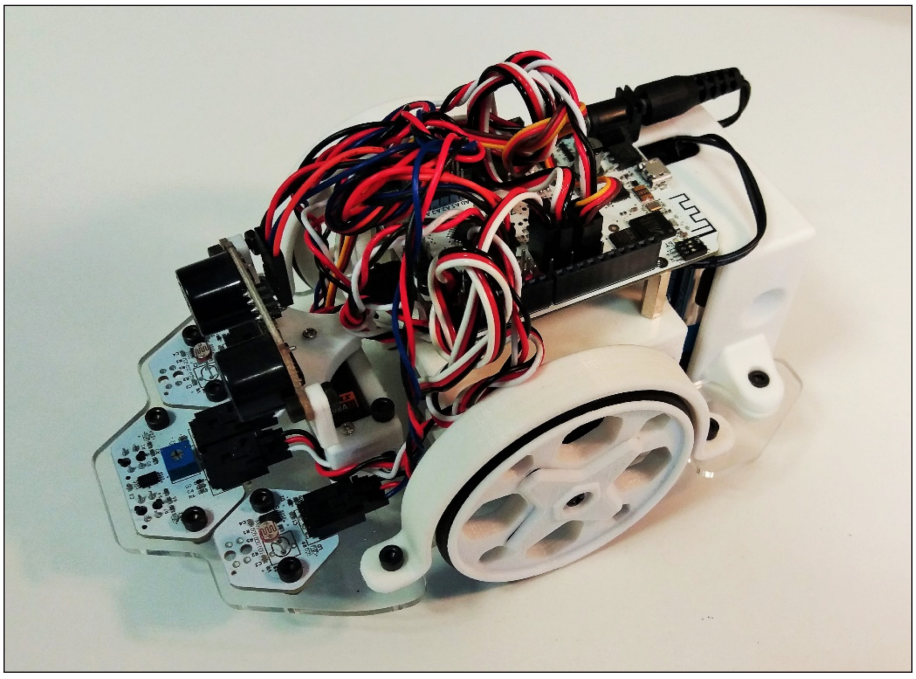

Figura 2. Robot Printbot Evolution de BQ montado

En cuanto a sensores dispone de diversas placas, denominadas ZUMbloq, que incorporan:

- 2 sensores de infrarrojo (IR), ubicados en la parte delantera para la aplicación de un robot sigue líneas.
- 2 sensores de luz tipo LDR, para seguir o evitar fuentes de luz.
- 1 sensor de ultrasonidos, para medir distancias a objetos y obstáculos.

En cuento a los actuadores cuenta con:

- 2 servomotores de rotación continua, acoplados a las 2 ruedas para proporcionar la tracción y el direccionamiento.
- 1 servomotor de 180º para direccionar el sensor de ultrasonidos.
- 1 zumbador para emitir sonidos.

Originalmente el Printbot Evolution emplea la placa controladora BQ Zum Core, compatible con Arduino. Esta placa se basa en un microcontrolador ATmega328P de 8 bits. Como características interesantes se puede destacar:

- Comunicación Bluetooth.
- Fácil conexionado de sensores y actuadores gracias a los conectores que incorpora.
- Fuente de alimentación de 3.2A, que permite alimentar directamente los servomotores.
- Interruptor para controlar la alimentación de los sensores y actuadores. Esto resulta interesante de cara a apagar estos elementos durante la programación.

Con el objetivo de poder emplear como placa controladora el Arduino Zero, por sus prestaciones más avanzadas, se han realizado algunos cambios en cuanto a componentes y proceso de ensamblado para poder integrarla con el Printbot Evolution. Estas modificaciones se enumeran a continuación:

1. Se ha sustituido la placa BQ ZUM Core por Arduino Zero. Al tener el mismo factor de forma esto no ha supuesto un problema a la hora de poder montarla en la estructura.
2. Se han añadido dos fuentes de alimentación que emplean pilas en lugar de una única. Una fuente de 9V alimenta la placa del Arduino Zero y

otra de 6V alimenta los servomotores, para adaptarse al nivel de tensión requerido por éstos.

3. Se ha añadido la placa Servo Shield de 16 canales PWM de 12 bits y comunicación I2C de Adafruit[2] (ver Figura 3). Esta placa se emplea para poder controlar los servomotores mediante una alimentación independiente. Además, permite ahorrar pines de E/S ya que se programa mediante I2C con tan sólo dos líneas. Esto permitiría añadir un gran número de servomotores adicionales si fuera necesario. Además, evita problemas de compatibilidad de tensiones ya que el Arduino Zero proporciona salidas de 3.3V, y evita cierto tipo de perturbaciones que se pueden producir al usar las salidas PWM, como por ejemplo *glitches*.
4. Se ha añadido la placa Sensor Shield V5[3], que permite un fácil conexionado de los sensores.
5. Se ha modificado la parte del porta pilas original, cortando la pieza que lo alojaba y añadiendo una placa donde montar los porta pilas para la pila de 9 V y las 4 pilas AAA que proporcionan los 6V.

Ambas placas al ser tipo *shield* y tener el mismo factor de forma que Arduino Zero son fácilmente conectables y proporcionan un montaje muy compacto.

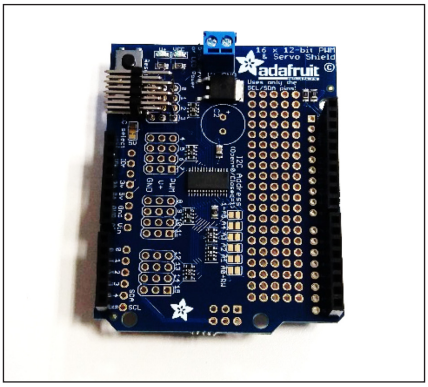

Figura 3. Servo Shield con 16 canales de 12 bits PWM y comunicación I2C (Adafruit)

El proceso de construcción del Printbot Evolution queda perfectamente detallado en las instrucciones de montaje que se proporcionan por parte del fabricante

2 https://www.adafruit.com/products/1411
3 https://www.electrohobby.es/es/shield/40-arduino-sensor-shield-v5.html

BQ[4]. El resultado final del ensamblaje de las diferentes partes y añadiendo las citadas modificaciones se puede ver en la Figura 4.

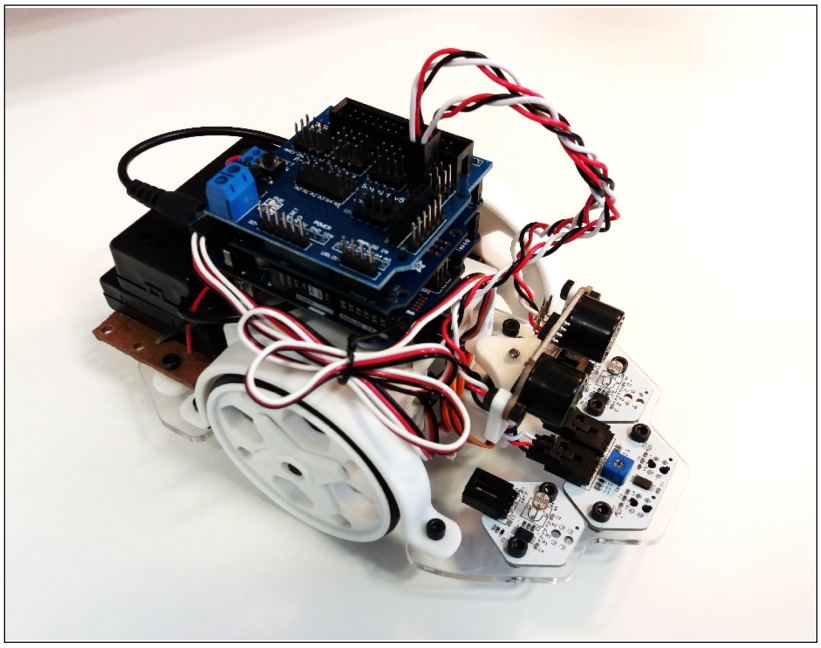

Figura 4. Resultado final de la construcción del robot móvil.

En muchos casos los sensores y actuadores que se emplean en robótica requieren una alimentación de 5V. En muchas placas de control, como el Arduino Uno o el ZUM Core de BQ, los microcontroladores y sus entradas y salidas digitales también emplean ese nivel de tensión. En el caso del Arduino Zero, al emplear tensiones de 3.3V en sus pines de E/S es necesario tener precaución para no sobrepasar esa tensión, ya que podría causar daños en el microcontrolador. En todo caso, en la actualidad existen muchos dispositivos electrónicos compatibles con estos dos niveles de tensión, no suponiendo ningún problema para el conexionado en estos casos.

Una de las limitaciones implícitas al uso de la placa Arduino Zero es que el regulador de tensión que proporciona la alimentación adicional de 5V puede sólo proporcionar una corriente continua de hasta 0.8A cuando se alimenta a través del conector de 2.1mm. Este nivel de corriente es todavía inferior si se emplea el

4 http://diwo.bq.com/montaje-del-printbot-evolution/

puerto USB ya que emplea una protección ante sobrecorrientes superiores a 0.5A mediante fusible reseteable. Este nivel de corriente resulta insuficiente cuando se conectan diversos servomotores por su elevado consumo. Además, se producen picos de corriente durante los transitorios de arranque y paro que pueden generar el reset del microcontrolador y el consiguiente reinicio del programa debido a la caída momentánea de la tensión de alimentación. Para subsanar esta limitación es necesario conectar el pin de alimentación de los servomotores a una fuente de alimentación externa que proporcione una tensión comprendida entre 4.8 y 6V, según sus especificaciones, en lugar de emplear el pin de alimentación de 5V de la placa Arduino Zero. El terminal de 0 V de la fuente externa y el de la placa Arduino Zero sí que deben unirse para proporcionar una referencia común. Esto justifica el uso de una doble fuente de alimentación como se ha descrito en las modificaciones. En el robot Printbot Evolution de BQ se subsana esta limitación mediante el uso de la placa Zum Core, compatible con Arduino, ya que incorpora un regulador de tensión de 5 V que puede proporcionar hasta 3.2A. Estás limitación quedan subsanadas en el diseño realizado mediante el uso de la placa Servo Shield de 16 canales PWM de 12 bits y comunicación I2C de Adafruit, citada previamente en esta sección.

4
INTRODUCCIÓN A ESTRUCTURA DE COMPUTADORES

El tema introductorio de estructura de computadores es el primer tema al que se enfrentan los estudiantes de la asignatura y tiene como objetivo principal el establecimiento de una base común sobre la que desarrollar el resto de la asignatura. Este tema ofrecerá una visión macroscópica de lo que, con más detalle, se estudiará en los temas posteriores y que conforma el denominado computador digital. La arquitectura clásica propuesta por von Neumann servirá para introducir brevemente los componentes fundamentales de un computador, así como las estructuras básicas utilizadas en su interconexión.

Por ser este un capítulo introductorio, las sesiones correspondientes del laboratorio se dedicarán a que el estudiante se familiarice con la plataforma Arduino Zero y su entorno de desarrollo.

El concepto de arquitectura von Neumann y sus diferencias con la arquitectura Harvard servirán como elemento conductor de las sesiones de prácticas 3 y 4 correspondientes a este primer tema, presentando la plataforma Arduino Zero como ejemplo de arquitectura Harvard.

Este tema, por lo tanto, se organizará de la siguiente manera. Primero, se describirán las características y detalles técnicos de la placa, incluyendo aspectos de procesado, depurado, entrada/salida, etc. Después, se analizará la diferencia entre las arquitecturas von Neumann y Harvard para finalmente pasar a describir el Arduino Zero como ejemplo de plataforma Harvard híbrida.

4.1. LA PLATAFORMA ARDUINO ZERO

La plataforma Arduino Zero[5] ha sido específicamente diseñada para proyectos del Internet de las Cosas (*Internet of Things*, IoT), aumentando su eficiencia para reducir el consumo de energía y ofreciendo una gran flexibilidad en cuanto a las interfaces periféricas. Esta plataforma incluye un microcontrolador Atmel (SAM D21) equipado con un procesador ARM Cortex M0+ de 32 bits, que trabaja a una frecuencia máxima de 48MHz. Una de las características más relevantes del microcontrolador de Atmel es su depurador empotrado EDGB que ofrece una interfaz de depuración que no requiere de ningún otro hardware adicional. El EDBG también ofrece un puerto COM virtual que puede ser utilizado para la programación del dispositivo y el *bootloader*.

4.1.1. EL MICROCONTROLADOR DE ATMEL

El microcontrolador de Atmel SMART SAM D21 proporciona las siguientes características [Atm16]. Una memoria Flash integrada programable, un controlador DMA (*Direct Memory Access*) de 12 canales, 12 canales de *Event System*, un controlador programable de interrupciones, hasta 52 pines programables de E/S, reloj y calendario de 32 bits, un máximo de 5 temporizadores de 16 bits y 3 de 24 bits. También dispone de un USB 2.0, hasta 6 módulos de comunicación serie (SERCOM) cada uno de los cuales se puede configurar para trabajar como una USART, UART, SPI, o I^2C de hasta 3.4MHz, entre otros, así como interfaz I^2S de dos canales. La tabla 4.1 resume todas estas características técnicas:

Tabla 1. Aspectos más relevantes de la configuración del microcontrolador Atmel SAM D21G18A.

Microcontrolador	ATSAMD21G18A, 32-bit ARM Cortex M0+
Tensión de alimentación	3.3V
Pines	48
Pines de E/S de propósito general	38
Flash	256KB
SRAM	32KB
Instancias del temporizador/contador (TC)	3
Canales de DMA	12

5 https://www.arduino.cc/en/Main/ArduinoBoardZero

Microcontrolador	ATSAMD21G18A, 32-bit ARM Cortex M0+
Interfaz USB	1
Instancias de Interfaz de comunicación serie (SERCOM)	6
Interfaz I²S	1
Canales de conversión analógico-digital (ADC)	14
Comparadores analógicos (AC)	2
Canales de conversión digital-analógico (DAC)	1
Contador de Tiempo Real (RTC)	Sí
Líneas de interrupciones externas	16
Frecuencia máxima de la CPU	48MHz
Canales Event System	12

4.1.2. EL PROCESADOR Y LA ARQUITECTURA ARM

La placa Arduino Zero y el microcontrolador SAM D21 que ésta integra disponen de un procesador ARM Cortex M0+ [ARM12] que está basado en una arquitectura ARMv6 y un juego de instrucciones Thumb-2. El procesador Cortex M0+ permite la configuración de aspectos como el *endianess* de los datos, el ancho de la instrucción capturada (16 o 32 bits), multiplicador de 32 bits rápido, etc. Se trata de un procesador multietapa de 32 bits y de tipo RISC. Además, es un procesador altamente eficiente en cuanto a consumo de energía, considerándose, por lo tanto, un procesador de bajo consumo energético. Este es uno de los motivos que hacen que la placa Arduino Zero esté especialmente pensada para aplicaciones relacionadas con IoT.

Los procesadores ARM emplean una arquitectura *load/store*, en la que las instrucciones se dividen en dos categorías:

1. Instrucciones de acceso a memoria (como por ejemplo transferir datos de registro a memoria y viceversa).
2. Instrucciones de operación con la ALU que tienen lugar entre registros.

Los registros de que dispone el procesador se describen en la Tabla 2.

Tabla 2. Registros del procesador ARM Cortex Mo+

Nombre	Descripción
R0-R12	Registros de propósito general para las operaciones con datos.
MSP (R13) PSP (R13)	El registro R13 es el *Stack Pointer* o puntero a pila (SP). El registro CONTROL indica qué puntero a pila utilizar: el *Main Stack Pointer* (MSP) o el *Process Stack Pointer* (PSP).
LR (R14)	El *Link Register* (LR) es el registro 14. Este registro almacena la información para el retorno de subrutinas, llamada a funciones y excepciones.
PC (R15)	El *Program Counter* (PC) es el registro R15. Este registro almacena la dirección actual de la instrucción a ejecutar.
PSR	El *Program Status Register* (PSR) combina los registros de estado: Application Program Status Register (APSR), Interrupt Program Status Register (IPSR), Execution Program Status Register (EPSR) y Estos registros proporcionan diferentes vistas del PSR.
PRIMASK	El registro PRIMASK previene la activación de todas las excepciones con prioridad configurable.
CONTROL	El registro CONTROL controla la pila utilizada y, opcionalmente, el nivel de privilegios del código cuando el procesador está en el modo *Thread*.

4.1.3. EL DEPURADOR EDBG

La placa Arduino Zero integra el depurador empotrado de Atmel (EDBG) [Atm14a]. Aunque se describirá con más detalle el proceso de depuración en el Anexo B, en la presente sección se describirán los aspectos más importantes de este hardware. El EDBG nos permite depurar el software que corre en la placa Arduino sin necesidad de recurrir a un depurador externo. El depurador aparecerá enumerado como un dispositivo USB compuesto, con dos interfaces para cada rol que puede desempeñar.

La placa Arduino no ofrece la posibilidad de depurar el código, de manera estándar o como lo realizamos cuando depuramos software en nuestro propio ordenador. Para ello y, dado que trabajaremos desde una arquitectura diferente

(la de nuestro ordenador) a la implementada por la placa Arduino, será necesario depurar de manera remota, para lo cual se lleva a cabo mediante un intercambio de mensajes a través de la clase *Serial*, como describiremos más adelante, entre nuestro ordenador y la placa.

4.1.4. ALIMENTACIÓN DE LA PLACA

La placa Arduino Zero puede alimentarse mediante una de las siguientes opciones:

1. A través de uno de los dos conectores USB: el nativo (conectado al SAMD21) o el de programación (conectado al chip EDBG).
2. Mediante una fuente externa tipo batería, fuente de alimentación o adaptador de red, empleando el conector de 2.1mm. También se puede emplear los pines VIN y GND del conector de alimentación para tal propósito. La tensión de alimentación deberá estar entre 6 y 20V, aunque se recomienda que esté en el rango entre 7 y 12V. Las tensiones de 5 V y 3.3V se generan a partir de un regulador conmutado y un regulador lineal, respectivamente, conectados en cascada como se muestra en la Figura 5.

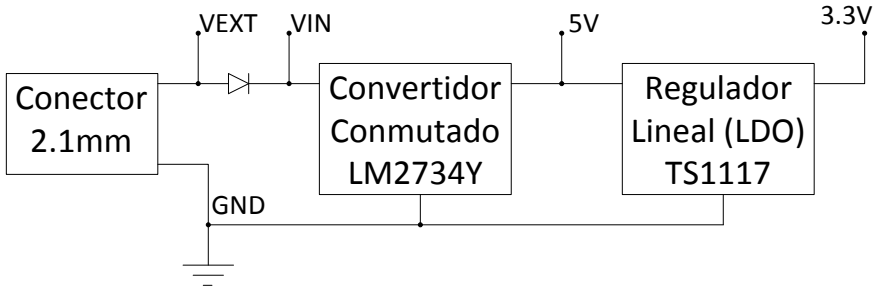

Figura 5. Diagrama de bloques de la alimentación de la placa Arduino Zero mediante alimentación externa

Para los pines de E/S, hay que tener presente que no debe aplicarse a estos pines una tensión superior a 3.3V. Esto es importante porque muchos otros modelos de placas Arduino empelan tensiones de 5V. La descripción de los diferentes pines que componen el conector de alimentación (*Power*) ser resume en la Tabla 3.

Tabla 3. Conector de alimentación

VIN	Tensión de entrada para la placa cuando se emplea una fuente externa de alimentación en lugar de los conectores micro USB. La placa se puede alimentar tanto a través del conector de 2.1mm o directamente a través de este pin del conector de alimentación.
GND	Referencia negativa de los diferentes pines de alimentación. Es común al resto de pines de alimentación de la placa.
5V	Salida del regulador conmutado de 5V. Aplicar tensión a través de los pines de 5V o 3.3V evita el paso a través del regulador lo que puede terminar dañando la placa al no existir protecciones de limitación de corriente.
3.3V	Salida del regulador lineal de 3.3V. La corriente máxima de salida es de 800mA. Este regulador es el que alimenta al microcontrolador SAMD21.
RESET	Señal de *reset* del micro. Permite emplear el botón de *reset* en otras placas tipo *shield* que se conecten a la placa Arduino Zero.
IOREF	Tensión de referencia con la que el microcontrolador trabajará. Así, los *shields* o placas auxiliares que han sido diseñadas adecuadamente pueden ver a qué tensión trabaja la placa y adaptarse a ésta.

4.1.5. EL SISTEMA DE MEMORIA

El esquema de memoria disponible con la placa Arduino Zero nos permitirá experimentar con diferentes tecnologías de memoria. Así, el microcontrolador SAMD21 cuenta con 256KB de memoria Flash, 32KB de memoria SRAM y hasta 16KB de EEPROM (por emulación). La Tabla 4 describe el mapa de memoria para el microprocesador SAMD21, que se verá en más detalle en el apartado 5 Memoria.

Tabla 4. Mapa de memoria física para el microprocesador SAMD21

Memoria	Dirección de inicio	Tamaño
Flash interna	0x00000000	256KB
Sección RWW interna	0x00400000	-
SRAM interna	0x20000000	32KB
Bridge periférico A	0x40000000	64KB

Memoria	Dirección de inicio	Tamaño
Bridge periférico B	0x41000000	64KB
Bridge periférico C	0x42000000	64KB

4.1.6. ENTRADA Y SALIDA

La entrada y salida (E/S) de la placa Arduino es especialmente importante porque no sólo nos permitirá explorar conceptos de la arquitectura, sino que determinará el tipo de aplicaciones que podamos desarrollar.

El Arduino Zero cuenta con 20 pines de E/S de propósito general. Como ya se ha comentado anteriormente, estos pines trabajan a un voltaje de 3.3V. Cada pin puede proporcionar o absorber un máximo de 7mA de corriente y tiene una resistencia (desconectada por defecto) de 20-50kΩ.

Algunos de los pines tienen asignada una funcionalidad específica:

- Serie: 0 (RX) y 1 (TX). Utilizado para el envío (RX) y la transmisión (TX) de datos serie, estándar o TTL.
- Interrupciones externas: Disponible en todos los pines menos en el pin 4.
- Salida analógica, DAC: A0. Proporciona una salida de 10 bits de voltaje con la función analogWrite( ).
- PWM: 3, 4, 5, 6, 8, 9, 10, 11, 12 y 13. Proporciona una salida PWM de 8 bits con la función analogWrite( ).
- SPI: SS, MOSI, MISO, SCK. Localizada en la cabecera ISCP sólo soporta comunicaciones SPI utilizando dicha librería.
- LED: 13. Hay un LED integrado en la placa y controlado por el pin 13. Cuando el pin está en valor alto o *HIGH*, el LED está encendido y cuando el pin está en valor bajo o *LOW*, está apagado.
- Entradas analógicas, ADC: 6 de los 20 pines de propósito general de E/S del Arduino Zero proporcionan entradas analógicas. Estas entradas se etiquetan desde la A0 a la A5. Cada una de estas entradas proporciona una resolución de hasta 12 bits (o dicho de otro modo, hasta 4096 valores diferentes).
- TWI: pines SDA y SCL. Estos pines soportan comunicación TWI a través de la librería WIRE.

Además, la placa Arduino Zero ofrece también otros pines:

- AREF: Este es el voltaje de referencia para las entradas analógicas. Se utiliza con la función analogReference( ).
- Reset: Al poner esta línea a nivel *LOW* (bajo) el microcontrolador se reseteará. Este pin se utiliza normalmente para añadir un botón de *reset* a los *shields*.

4.1.7. LA PROGRAMACIÓN

Lo más relevante del proceso de programación de la placa Arduino Zero es que éste difiere del que se emplea para los microcontroladores AVR de otras placas Arduino y Genuino. Mientras que otras placas utilizan avrdude para la carga de los sketchs, la placa Arduino Zero utiliza bossac[6] y openOCD[7] para el puerto de programación.

La Figura 6 indican los dos puertos USB que pueden ser utilizados para la programación de esta placa. Aunque ambos puertos pueden utilizarse indistintamente para la programación de la placa, el uso del puerto de programación se prefiere al del puerto nativo, por la manera en la que se gestiona el borrado del chip. Para usar el puerto de programación, simplemente habrá que indicar en el IDE de Arduino que se empleará dicho puerto. En cuanto al puerto nativo, éste está conectado directamente al microprocesador SAMD21.

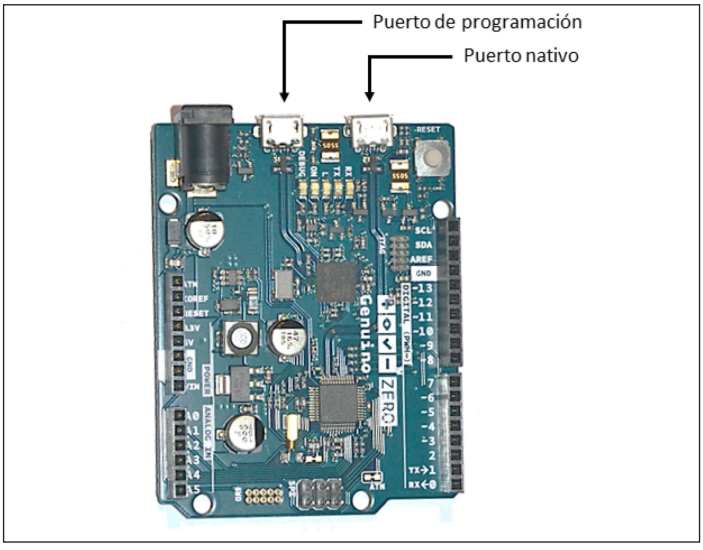

Figura 6. Puertos USB de programación de la placa Arduino Zero

6 https://sourceforge.net/projects/b-o-s-s-a/
7 http://openocd.org/

4.1.8. EL PROCESO DE ARRANQUE

El microprocesador SAMD21 viene con un bootloader precargado, conocido como SAM-BA [Atm14b]. Esta herramienta permite la programación del microcontrolador utilizando un programador externo. Normalmente, los bootloaders ocupan espacio físico de la memoria Flash, sin embargo, el SAM-BA al estar grabada en la memoria ROM del chip, directamente desde fábrica, no ocupa ningún espacio adicional de la memoria Flash.

4.2. TIPOS DE ARQUITECTURA: VON NEUMANN Y HARVARD

Antes de adentrarnos en la explicación de cada una de estas arquitecturas y establecer las principales diferencias que existen entre ellas, convendría ir un paso más atrás y plantearnos qué es un computador, qué es la arquitectura de un computador y por qué existen distintas arquitecturas.

4.2.1. ¿QUÉ ES UN COMPUTADOR?

Un computador no es más que una máquina destinada a procesar información, entendiendo por procesamiento toda aquella manipulación o transformación a la que se somete la información para resolver un problema determinado.

Actualmente, los computadores utilizan tecnología mayoritariamente electrónica para su construcción y pueden representar la información de dos maneras bastante diferenciadas, lo que nos lleva a la distinción entre computadores digitales y calculadores analógicos. En el caso que nos ocupa sólo nos interesan los computadores digitales en los que la información se representa mediante un sistema digital de tipo binario.

4.2.2. ARQUITECTURA DE UN COMPUTADOR

La arquitectura de un computador, básicamente, define su comportamiento funcional. Podemos definir la arquitectura de un computador como el diseño conceptual y la estructura operacional fundamental de un computador, es decir, es un modelo y una descripción funcional de los requerimientos y las implementaciones de diseño para varias partes del computador, mostrando especial interés en la forma en que la Unidad Central de Proceso (CPU) trabaja internamente y accede a las distintas direcciones de memoria. Podríamos decir también que es la forma de interconectar componentes hardware para crear computadores según unos requerimientos de funcionalidad, rendimiento y coste.

La arquitectura de un computador muestra la situación de sus componentes y permite determinar las posibilidades para que un sistema informático, con una determinada configuración, pueda realizar las operaciones para las que se va a utilizar. La arquitectura básica de cualquier computador está formada por cinco componentes básicos: procesador, memoria principal, disco duro, dispositivos de entrada/salida y software.

4.2.2.1. ARQUITECTURA VON NEUMANN

La arquitectura Von Neumann fue establecida por John Von Neumann en 1945. Un computador con esta estructura es capaz de ejecutar una serie de instrucciones elementales (instrucciones máquina) almacenadas en una memoria principal y está compuesta por las siguientes unidades, tal y como podemos ver en la Figura 7:

- Memoria Principal
- Unidad Aritmética
- Unidad de Control
- Unidad de Entrada/Salida

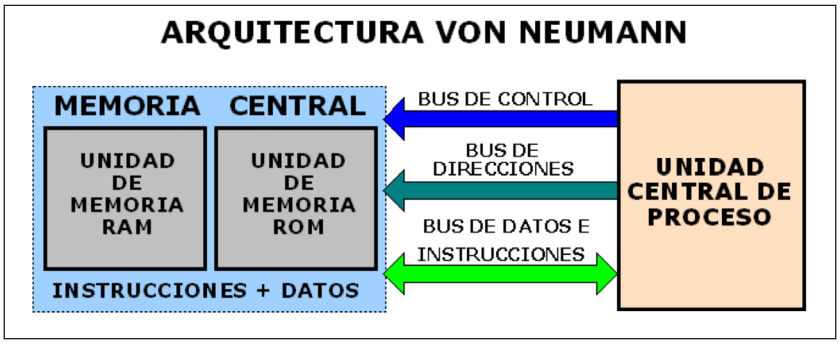

Figura 7. Arquitectura von Neumann

A estas unidades habrá que añadir unos caminos o buses que permiten que los datos y las instrucciones circulen entre las distintas unidades del computador.

Esta arquitectura se caracteriza, básicamente, porque los procesadores poseen el mismo dispositivo de almacenamiento tanto para los datos como para las instrucciones. Al ser almacenados en el mismo formato dentro de la misma memoria, utilizan un único bus de datos para comunicarse con la CPU, lo que hace que la

utilización de la memoria sea eficiente en cuanto a uso de recursos, pero presenta una ambigüedad para reconocer los datos y hace que las lecturas de los datos y las instrucciones sea secuencial, lo que puede provocar que sea más lento que otro tipo de arquitecturas.

4.2.2.2. ARQUITECTURA HARVARD

La arquitectura Harvard, representada en la Figura 8, fue establecida en 1947 y debe su nombre a la computadora Harvard Mark I basada en relés. Se caracteriza por tener dos memorias principales: una para almacenar las instrucciones y otra para almacenar los datos.

La ventaja de esta arquitectura es que permite leer en paralelo datos e instrucciones, por lo que puede ser más rápida que la arquitectura von Neumann. El inconveniente es que, en muchos casos, no se aprovecha bien la memoria disponible, ya que puede ser necesario espacio adicional en un tipo de memoria mientras que de la otra puede haber espacio disponible.

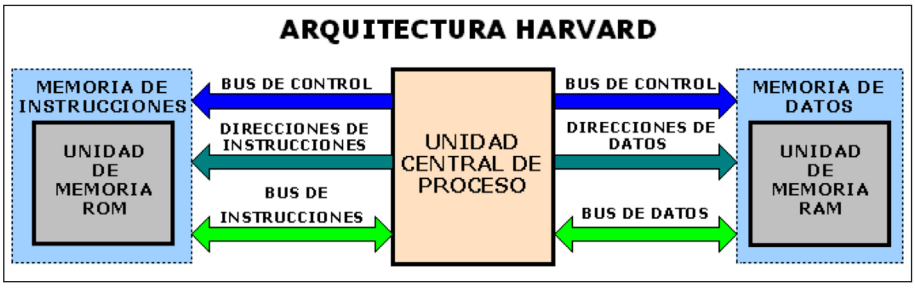

Figura 8. Arquitectura Harvard

4.2.2.3. ARQUITECTURA VON NEUMANN VS. ARQUITECTURA HARVARD

La principal característica que diferencia estos dos tipos de arquitectura, como ya hemos visto, es la memoria que, en la arquitectura von Neumann es única y en la Harvard tiene dos diferentes una para los datos y otra para las instrucciones.

También podemos establecer otro tipo de diferencias:

Arquitectura Von Neumann	Arquitectura Harvard
Datos e instrucciones en una misma memoria de lectura/escritura.	Datos e instrucciones en memorias caché separadas.
Se puede diferenciar entre datos e instrucciones al examinar una posición de memoria (*Location*).	No es necesario discriminar el tipo de dato con el que se está trabajando.
Los contenidos de memoria son direccionados por su ubicación sin importar el tipo de datos que contengan.	Los contenidos de las memorias se direccionan dependiendo del tipo de dato que sea.
Ejecución secuencial de lectura de la memoria.	Ejecución paralela de lectura de instrucciones y datos

4.3. ARDUINO ZERO COMO EJEMPLO DE ARQUITECTURA HARVARD

La característica fundamental de la arquitectura Harvard es que tiene una memoria para instrucciones y otra para datos. En el caso de Arduino Zero, las instrucciones se cargan en la memoria Flash del microcontrolador mientras que los datos estarán en la memoria SRAM. Al ser la memoria Flash de tipo no volátil el programa no se pierde al quitar la alimentación de la placa.

En la memoria de programa, por lo tanto, residen las instrucciones en lenguaje máquina que serán ejecutadas por el procesador. A esta memoria se hace referencia algunas veces como memoria ROM (*read-only memory*) por herencia de lo que ocurría anteriormente cuando las memorias no volátiles eran casi exclusivamente solo de lectura. Este ya no es el caso, pero es todavía común encontrar referencias a la memoria de programa como memoria ROM [Whe11].

Por su parte, la memoria de datos es aquella en la que el microcontrolador almacenará las variables utilizadas durante la ejecución del programa. Esta memoria se implementa utilizando como tecnología memorias estáticas de acceso aleatorio, o SRAM, que mantienen la información mientras que la placa está alimentada, perdiéndose una vez que se desconecta de la fuente de alimentación. Una vez que la memoria deja de estar alimentada no puede asumirse que sus celdas de memoria tomarán un valor específico (por ejemplo, ceros) [Whe11].

5
MEMORIA

El sistema de memoria es el encargado de almacenar información, bien de manera permanente o temporal. Se trata pues de un elemento esencial porque en ella se almacenarán tanto los datos como las instrucciones del programa a ejecutar. Se puede por lo tanto afirmar que la memoria del computador es un elemento fundamental del mismo. De hecho, sin la memoria, el computador no podría ni arrancar.

Las operaciones básicas sobre la memoria son la de escritura (o almacenamiento) y lectura. En ambos casos es necesario suministrar una dirección de memoria donde almacenar la información o desde dónde leerla. A nivel de sistema, la memoria se ve como una caja negra, a la que hay que suministrar una dirección y unas señales de control que indicarán el tipo de operación a realizar (lectura o escritura). Además, si es una operación de escritura es necesario suministrar el dato a escribir.

5.1. GESTIÓN DE LA MEMORIA EN ARDUINO ZERO

Al contrario de lo que ocurre con los computadores de propósito general, los microcontroladores y dispositivos empotrados tipo Arduino, están pensados para estar dedicados a una única tarea. El objetivo en este tipo de plataformas es, por lo tanto, optimizar la arquitectura para el desarrollo de esa tarea. En este contexto, la arquitectura Harvard encuentra una aplicación idónea, por

cuanto la separación de programa y datos, con buses dedicados, permitirá una ejecución más optimizada.

El estado de un programa en ejecución vendrá determinado, en cada momento, por el contenido de la memoria y los registros y, más concretamente, por el contenido de cada una de las secciones en las que se divide la memoria de un programa en ejecució, como son:

- La sección de texto o *.text,* donde se localiza el código en ejecución. Dentro de esa misma sección, habrá otra subsección que puede contener textos fijos.
- La sección de datos o *.data* para los datos de solo lectura o variables globales.
- La sección de variables estáticas o globales no inicializadas o *.bss.*
- La pila o *stack*, para las variables automáticas y el soporte a la llamada a procedimientos.
- El montículo o *heap* para la gestión dinámica de la memoria.

Esas secciones de memoria genéricas de cualquier programa en ejecución se van a ubicar físicamente en diferentes tipos de memoria. En el caso del Arduino Zero, en las memorias Flash y SRAM internas. Generalmente, una plataforma Arduino cuenta con tres tipos de memoria, como son:

- La memoria Flash para las instrucciones y datos de solo lectura.
- La memoria SRAM para los datos que se perderán una vez que la placa se desconecte de la alimentación.
- La memoria EEPROM,para aquellos datos que necesiten persistencia tras la desconexión de la alimentación o un *reset*. El Arduino Zero no dispone de memoria EEPROM.

El Arduino Zero es también especial en cuanto al espacio de direccionamiento que implementa. Como se puede observar en la Figura 9[8] hay un único espacio de direccionamiento que engloba a la memoria Flash y la SRAM y que va de la dirección 0x00000000 a la 0xFFFFFFFF. Por este motivo, el acceso a la memoria Flash ya no requiere el uso de PROGMEN, sino que ésta se puede referenciar directamente.

8 Fuente: http://www.sumidacrossing.org/Musings/files/160606_Memory_and_the_Arduino.php

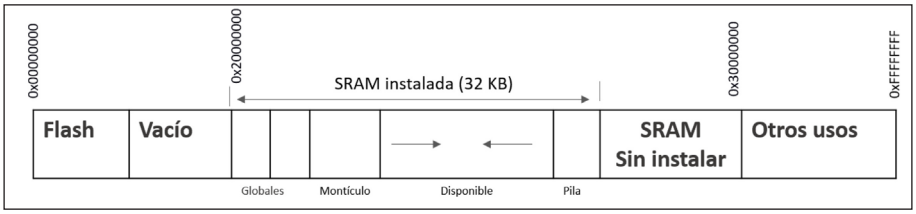

Figura 9. Esquema de memoria del Arduino Zero

La Figura 9 también nos permite apreciar dónde se mapea cada una de las secciones de memoria y en qué memoria física se contiene (Flash o SRAM). Así, podemos ver como por ejemplo, la pila está en la SRAM y crece en sentido descendiente, en cuanto a posición de memoria, a partir de la dirección 0x20007FFF.

La relación entre el montículo y la pila es que ambas comparten la misma región de memoria y crecen de manera opuesta, pudiendo sus extremos llegar a encontrarse. La Figura 10[9] representa como en un contexto normal de ejecución habrá un espacio disponible entre la cima de la pila y el montículo. Sin embargo, cuando una de las dos crece descontroladamente y ambos extremos se encuentran, el programa responderá de manera inesperada, pero tampoco sin indicación expresa de lo ocurrido.

La pila, como su nombre indica, implementa un mecanismo LIFO (*last in, first out*) de tal forma que los últimos datos introducidos (utilizando la instrucción *push*) en la pila serán los primeros en ser extraídos (utilizando la instrucción *pop*).

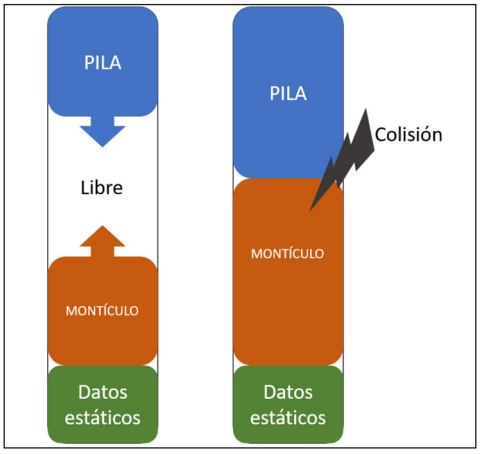

Figura 10. Relación entre el montículo y la pila

9 Fuente: https://learn.adafruit.com/memories-of-an-arduino/optimizing-sram

Por otra parte, la sección .bss dedicada a las variables estáticas o globales no inicializadas se habrá inicializado al valor cero. Puede por lo tanto asumirse que las variables globales no inicializadas estarán inicializadas a cero, aunque no puede asumirse lo mismo para cualquier otro tipo de variable.

El último elemento de almacenamiento del que no hemos comentado aún nada son los registros internos del procesador. Aunque posteriormente se tratará otra serie de registros propios de la gestión de la comunicación con los dispositivos periféricos, en este momento nos centraremos en los registros que componen el nivel superior de la jerarquía de memoria de un computador.

El procesador Cortex M0+ del Arduino Zero, dispone en una amplia variedad de registros de propósito general, en concreto, 13 registros numerados de R0 a R12, de 32 bits y 3 registros de propósito específico, como son:

- PC: El contador de programa o *Program Counter*. Este registro se carga con la dirección de inicio de programa cuando se resetea o se alimenta el microcontrolador, y contiene la dirección que está siendo procesada durante la ejecución del programa.
- LR: El *Link Register* es el registro encargado de almacenar la dirección de retorno durante la llamada a un procedimiento, aunque también puede utilizarse para otros propósitos cuando no está siendo utilizado.
- SP: El *Stack Pointer* o puntero a pila, es el encargado de apuntar a la cabeza de la pila.

5.2. EL APPLICATION BINARY INTERFACE (ABI)

El ABI de una determinada arquitectura consiste en una colección de estándares que definen las reglas que, entre otros, tendrán que seguir los compiladores encargados de generar código objeto o binario para esa arquitectura. El motivo por el que incluimos una sección sobre el ABI en el capítulo de memoria es porque éste regula muchas cuestiones relativas al uso que un programa hace de la memoria de un computador. Ejemplos de ello son el alineamiento de los datos en la memoria o la llamada a procedimientos. Las reglas de alineamiento determinan en qué posición de memoria deberán escribir los datos dependiendo de su tipo y con el objetivo de minimizar el número de accesos a memoria que se necesita para recuperar un dato de memoria. En cuanto a la llamada a procedimientos, el ABI determina cómo debe llevarse a cabo el paso de argumentos y retorno de valores.

El ABI será específico de la arquitectura, en nuestro caso hablaremos del ABI para ARM, pero también del lenguaje, en nuestro caso el ABI de C++ pues el lenguaje de Arduino es una extensión de C/C++. El *Embedded* ABI o EABI recoge las reglas para la generación de código objeto para procesadores ARM, como es el caso del Cortex M0+, y será el que analizaremos para el estudio de dos cuestiones fundamentales del uso de la memoria como son las reglas de alineamiento y la llamada a procedimientos.

5.2.1. EL ENDIANNESS

Nos referimos por *endianness* al orden en que se guardan en memoria los bytes que componen una palabra. Hablaremos de una configuración *little endian* cuando el byte más significativo se guarda en las posición de memoria más baja y *big endian* cuando éste se guarda en la posición de memoria más alta.

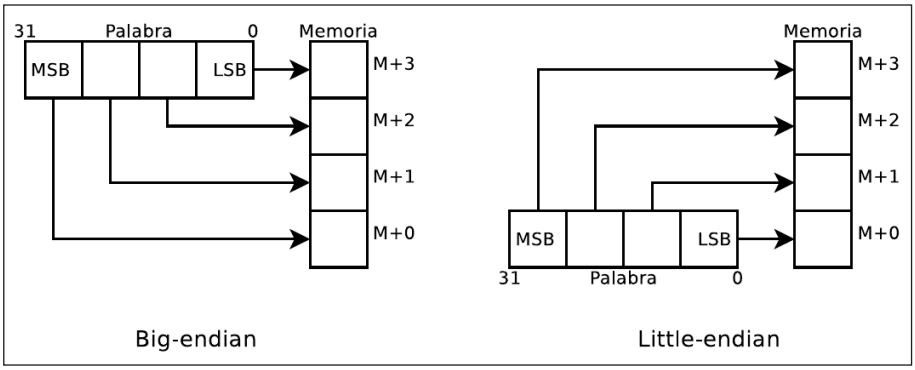

Figura 11. Tipos de *endianness*

La Figura 11 ilustra ambos casos. En primer lugar, puede observarse como en una configuración *little endian*, el byte más significativo (MSB, *Most-Significant Byte*), que incluye los bits 24-31 de la palabra se almacena en la posición de memoria mayor (M+3). En cambio, en la configuración *big endian*, el byte menos significativo (LSB, *Least-Significant Byte*), bits 0 a 7, se escriben en la posición de memoria más baja (M+0).

El EABI permite cualquiera de las dos configuraciones porque, a su vez, el procesador Cortex M0+ también permite cualquiera de ellas. La configuración *little endian*, además de ser la configuración más extendida, es la implementada por Arduino, pero es importante tener en cuenta que la otra configuración también es posible.

5.2.2. REGLAS DE ALINEAMIENTO

La memoria es un recurso limitado, más si cabe en dispositivos empotrados como Arduino, alguno de los cuales no superan los 2048 bytes de SRAM. Como programadores, la forma en la que escribimos código tiene un impacto en el mejor o peor uso que hagamos de este limitado recurso. Por lo tanto, conocer qué reglas sigue el compilador a la hora de decidir cómo se organizan los datos en memoria es fundamental para conseguir un uso óptimo.

Las reglas de alineamiento describen cómo, dependiendo del tipo de dato en cuestión, se escribirán las variables de más de un byte (variables tipo int, short, double, etc.) en la memoria, teniendo en cuenta el tamaño de palabra utilizado por la arquitectura y el tamaño de los buses (o canales por los que la información fluye entre las diferentes unidades de un computador y la memoria). El objetivo de las reglas de alineamiento no es únicamente optimizar el tamaño que los datos ocupan en la memoria sino, sobre todo, el número de accesos que es necesario realizar para leer un dato de la memoria. Así, por ejemplo, si tenemos un bus que nos permite transportar datos de 32 bits (o lo que es lo mismo, tenemos buses de 32 bits) y por otro lado queremos acceder al valor que en memoria tiene una variable de tipo entero, lo lógico es esperar que en una única lectura se pueda acceder a ese valor. No tendría sentido realizar dos accesos a memoria para leer una palabra de 32 bits. Figura 12 representa el caso en el que el acceso no estuviera alineado, así, para acceder a la palabra compuesta por el byte (i5,i6,i7,i8) sería necesario realizar dos accesos: el primero representado por la línea punteada naranja y el segundo por la línea punteada amarilla.

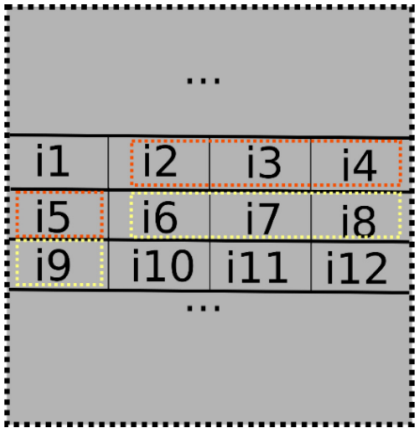

Figura 12. Simulación de un acceso no alineado a datos en

Parece obvio, en el caso de los enteros descritos en la Figura 12, que los accesos deben realizarse desde el comienzo de la palabra, de tal forma que en un único ciclo de lectura se haya accedido al dato completo. Sin embargo, cuando los datos no son del mismo tamaño que el de la palabra de memoria (en este caso, 32 bits), es necesario conocer esas reglas de alineamiento para saber en qué posición deberán ser escritos para minimizar así el número de accesos.

Estas reglas, detalladas en [ARM16], establecen que todos los tipos básicos se alinean a posiciones de memoria que sean múltiplos de su tamaño, de tal forma que el tipo short podrá alinearse al límite de la media palabra o múltiplo 2 bytes, el int al límite de la palabra o múltiplo de 4 bytes, etc. Es interesante ver lo que ocurre con los tipos compuestos, como por ejemplo los creados a partir de una estructura. Los tipos que internamente componen la estructura estarán alineados al de mayor tamaño y, a su vez, la estructura estará alineada a ese tamaño.

5.2.2.1. ALINEAMIENTO DE TIPOS COMPUESTOS

El lenguaje de C para Arduino admite las mismas formas que el propio lenguaje C para crear tipos compuestos: estructuras, vectores y uniones.

Las estructuras se definirán mediante la palabra reservada *struct* y permite crear un tipo compuesto a través de la agrupación de varios tipos básicos. En este caso, las reglas de alineamiento establecen que los tipos básicos que componen la estructura se alinearán al tamaño del tipo de mayor tamaño y, a su vez, la estructura se alineará a un tamaño múltiplo del tamaño del tipo de mayor tamaño.

El otro tipo compuesto, los vectores, al componerse de una sucesión de un único tipo básico no plantea mucho problema, pues el tipo compuesto estará alineado a un múltiplo del tipo básico.

Finalmente, las uniones, definidas utilizando la palabra reservada *union* se diferencian de las estructuras en que todos los elementos básicos que lo componen se almacenan en la misma dirección de memoria por lo que el alineamiento de dicho tipo compuesto será similar al de las estructuras.

Para ilustrar la importancia de conocer estas reglas por el impacto que tienen sobre el uso de la memoria, analizaremos el código propuesto en [MFSR10].

```
struct MyData {
char c;
double d;
int s;
};

int main () {
char c;
struct MyData data[5];
...
```

En este código vemos como se ha declarado un vector de 5 elementos del tipo compuesto MyData. El alineamiento del vector será el de la estructura, ocupando después el vector en memoria, 5 veces dicho espacio. Para explicar cómo se lleva a cabo el proceso de alineamiento en memoria siguiendo las reglas establecidas por el ABI recurriremos a la ilustración de [MFSR10] en la Figura 13.

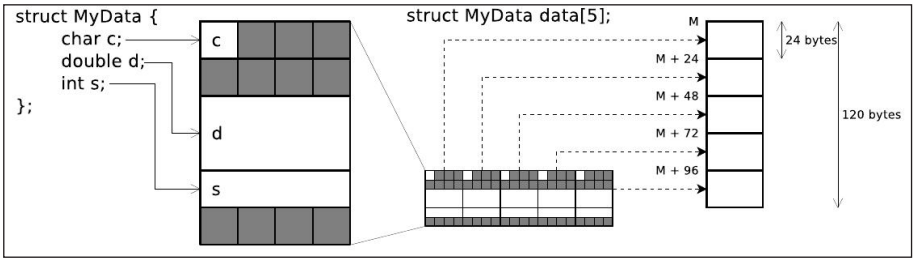

Figura 13. Alineamiento en memoria del tipo compuesto

Como se indica en la Figura 13, en primer lugar encontramos la declaración de la variable tipo char que, como indica el ABI puede estar alineado a posiciones de memoria múltiplo de 1 byte, por lo tanto puede escribirse en cualquier posición de memoria. Después, el siguiente tipo básico que forma la estructura es el double y, en este caso, al ser éste un tipo de 8 bytes, debe escribir alineado a una posición de memoria múltiplo de 8 bytes. Esto significa que es necesario utilizar 7 bytes de relleno (representado mediante las celdas coloreadas en gris) para alcanzar la siguiente posición de memoria múltiplo de 8. Finalmente, el último elemento es de tipo int y éste debe estar alineado a posiciones múltiplos de 4 bytes, pudiendo por lo tanto escribir a continuación de la variable d. Una vez alineados los tipos básicos de la estructura, habrá que alinear la estructura a un número múltiplo del tamaño del mayor tipo de la estructura, en este caso el double, por lo que habrá que utilizar 4 bytes de relleno para que la estructura quede alineada en memoria.

Como en nuestro código hemos declarado un vector de 5 elementos de este tipo compuesto, como se puede ver en la misma Ilustración 8, la región sombreada en gris representa los bytes de relleno y que por lo tanto están ocupando espacio inútilmente.

La mejor manera de comprender el verdadero impacto del alineamiento de datos es comparando ese mismo ejemplo con un código más optimizado. Si ahora la estructura se define de la siguiente manera:

```
struct MyData {
double d;
int s;
char c;
}
```

Siguiendo las reglas de alineamiento y utilizando la representación propuesta en [MFSR10], vemos en la Figura 14 como la cantidad de espacio de relleno (por lo tanto espacio desaprovechado) se ha reducido de manera muy significativa, pasando de ocupar 24 a 16 bytes en su versión optimizada.

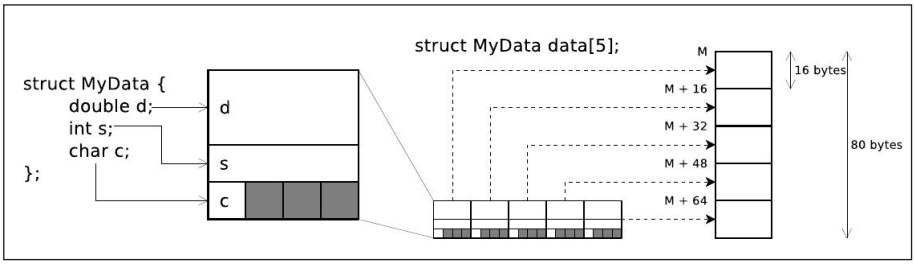

Figura 14. Optimización de la declaración del tipo compuesto y su alineamiento en memoria

Lo cierto es que aunque, como programadores, escribamos código no optimizado, el compilador se encargará de realizar este tipo de ajustes por nosotros.

5.2.3. LA LLAMADA A PROCEDIMIENTOS

El ABI también establece cómo debe llevarse a cabo la llamada a subrutinas o llamada a procedimientos y, más concretamente, cómo debe llevarse a cabo el paso de argumentos y retorno de valores.

Utilizando el código del *"Hola, Mundo"* disponible en el anexo C podemos observar como lo que se hace en cualquiera de las dos sentencias es una llamada a dos procedimientos: por un lado, una llamada al procedimiento begin para inicializar la comunicación con el puerto serie a una determinada velocidad y por otro, en el caso del print, para llevar a cabo la escritura a través del puerto serie.

```
void setup() {
Serial.begin(9600);
Serial.print("Hello, World");
}

void loop() {
}
```

Si observamos el código ensamblador generado, utilizando para ello la herramienta de depuración KDbg, como se puede ver en la Figura 15, identificamos como ambas tienen en común el uso de la instrucción bl (*branch and link*) justo antes de saltar a la dirección de memoria a partir de la cual se encuentra la primera instrucción del procedimiento a ejecutar.

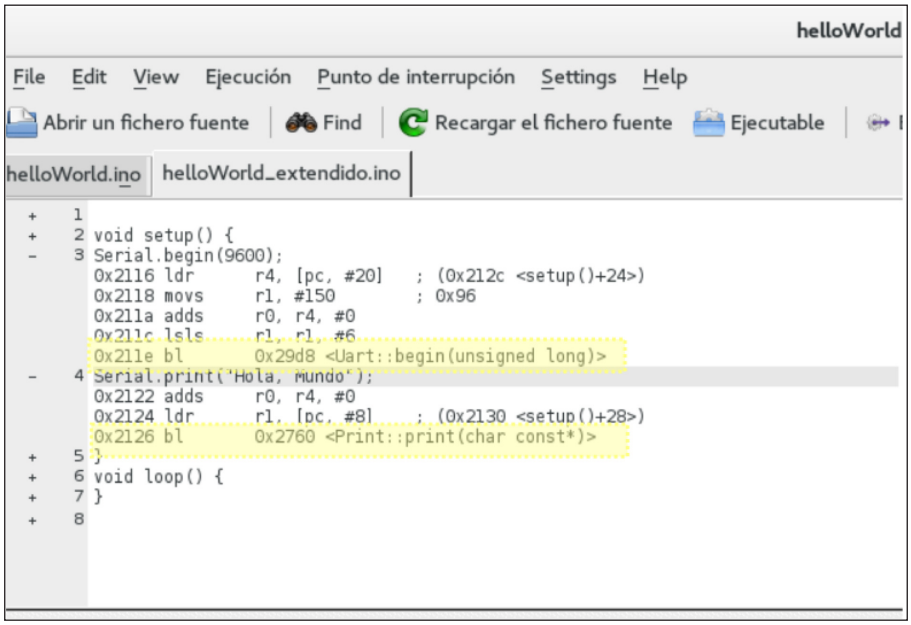

Figura 15. Ensamblador asociado a una llamada a procedimiento

La instrucción *bl* es la encargada de modificar el contador de programa o registro PC para que en lugar de continuar con la ejecución de instrucciones localizadas en posiciones contiguas de memoria, salte a una nueva posición, donde encontrará las instrucciones del procedimiento en cuestión. En este caso, primero ejecutará el procedimiento *begin* y después el *print.* Como se puede observar, la instrucción *bl* sólo recibe como operando la dirección de memoria donde se encuentra la primera instrucción del procedimiento.

5.2.3.1. PASO DE ARGUMENTOS

El ABI establece cómo habrá de llevarse a cabo el paso de argumentos dependiendo del número y tipo de argumentos. Siempre que sea posible se utilizarán los registros de propósito general R0, R1, R2, R3. Si estos registros no fueran suficientes para albergar el tipo o número de parámetros, se recurre al uso de la pila y memoria, para lo cual se sigue el algoritmo en tres etapas, descrito en [MFSR10], traducido del AAPCS [ARM15].

- La etapa de *inicialización* se realiza una sola vez antes de procesar los argumentos. Se utilizan un par de contadores que llevan la cuenta de cuál es el primer registro libre para la asignación de un argumento (SiguienteRegistro, que se inicializa a R0) y cuál es la siguiente posición de la pila libre para asignar argumentos (SiguienteArgEnPila, que se inicializa a SP). Si la función devuelve un resultado en memoria, la dirección del resultado se escribe en R0 y SiguienteRegistro pasa a ser R1.
- La etapa de *pre-padding* y e*xtensión de argumentos* se ejecuta por cada argumento después de la inicialización. Básicamente el objetivo de esta etapa es hacer que los argumentos ocupen un número entero de palabras.
    - Si se trata de un argumento compuesto y no se puede determinar el tamaño por la función, se copia en memoria y se reemplaza por un puntero a esta copia.
    - Si es de un tipo básico de menos de 4 bytes, se extiende a 4 bytes.
    - Si es de un tipo compuesto cuyo tamaño no es múltiplo de 4 bytes, se redondea a múltiplo de 4 bytes más cercano.

- Por último la etapa de *asignación de argumentos a registros y pila* también se ejecuta iterativamente por cada argumento.

- Si el argumento requiere 8 bytes asigna a SiguienteRegistro el siguiente registro par. Es decir los argumentos de 8 bytes se pueden asignar a R0 y R1 o a R2 y R3, pero no a R1 y R2.
- Si el tamaño del argumento no supera el tamaño de los registros disponibles para paso de argumentos, entonces se copia a registros empezando en SiguienteRegistro, y se incrementa SiguienteRegistro hasta el siguiente registro libre.
- Si no cabe en los registros disponibles pero SiguienteArgEnPila contiene sp, es decir, todavía no se ha asignado ningún argumento a la pila, entonces se divide el argumento entre registros (primera parte) y pila e incrementa SiguienteArgEnPila a la siguiente posición libre.
- En cualquier otro caso copia el argumento a SiguienteArgEnPila e incrementa SiguienteArgEnPila a la siguiente posición libre.

5.2.3.2. VALOR DE RETORNO

Al igual que ocurre con el paso de argumentos, el AAPCS establece el mecanismo que debe seguirse para el retorno de valores dependiendo del tipo y tamaño del valor a devolver. Siempre que sea posible, se establece el uso prioritario del registro R0 pero, cuando por tamaño, esto no es posible, se establece un mecanismo alternativo. El siguiente algoritmo, tal y como se describe en [MFSR10], resume lo que el AAPCS [ARM15] establece para el retorno de valores:

- Si el valor de retorno es de un tipo básico o compuesto de tamaño menor o igual a 4 bytes (char, short, int, long, float) entonces se retorna en R0. Si es necesario se expande a 4 bytes manteniendo el valor. Por ejemplo, un char con el valor -19 (oxed en hexadecimal) pasará de ocupar un byte a ocupar los 4 bytes de r0 (oxffffffed en hexadecimal). Para ello el compilador deberá utilizar la operación de extensión de signo. Los tipos compuestos pequeños se retornan en R0 sin extensión de ningún tipo. La disposición de los datos es la misma que tendría en memoria.
- Si el valor de retorno es de un tipo básico de 8 bytes (double, long long) entonces se retorna en R0 y R1.
- Si el valor de retorno es de un tipo compuesto de más de 4 bytes o de tamaño indeterminado, entonces el resultado se devolverá en un área de memoria (pila) determinada por el que llama. La dirección de esta área de memoria se pasa a la función como un argumento extra.

5.3. LA PILA

La pila es una región de memoria, en nuestro caso ubicada en la memoria SRAM, que comparte espacio con el montículo como se puede observar en la Figura 9 y cuya función es la de almacenar variables automáticas o temporales y también servir de soporte durante la llamada a procedimientos almacenando aquellos argumentos que no puedan ser pasados a través de los registros. Este espacio de memoria se comporta como un buffer FILO *first in last out*.

El hecho de que la pila y el montículo compartan la misma memoria puede dar lugar a que el programa se quede sin memoria cuando una de las dos secciones crece de manera no controlada. Ambas áreas comparten región de memoria y crecen desde los extremos opuestos. Así, cuando los extremos se encuentran se habrá agotado la memoria disponible, dando lugar a un comportamiento inesperado de nuestro programa. Para controlar el crecimiento de ambas áreas disponemos de dos punteros que referencian a la cabeza de la pila y del montículo. El puntero a pila o *stack pointer* (SP) es uno de los registros de propósito específico del procesador.

La pila juega un doble papel, por un lado almacenando las variables automáticas y por otro dando soporte al paso de argumentos cuando se lleva a cabo una llamada a procedimiento. Así, cuando se lleva a cabo una llamada a un procedimiento, el compilador se habrá encargado de añadir el código necesario para reservar el espacio necesario en la pila para albergar las variables automáticas que serán utilizadas durante dicho procedimiento. Esta reserva de espacio en la pila se lleva a cabo desplazando el puntero a pila o SP tantas posiciones como espacio ocupen dichas variables. Debe tenerse en cuenta que las reglas de alineamiento en memoria estudiadas anteriormente también aplican al almacenamiento de datos en la pila Ese espacio reservado se denomina marco de pila o *stack frame*. Esa región, además, estará delimitada por un puntero denominado *frame pointer* de tal forma que, por ejemplo, las variables automáticas o locales al método en ejecución estarán siempre comprendidas entre el *stack pointer* y el *frame pointer*.

El manejo de la pila se realiza mediante dos tipos de instrucciones: la instrucción push para insertar un elemento en la pila y la instrucción *pop* para sacar el último elemento de la pila. En el procesador Cortex Mo+ se implementa un modelo de pila *full-descending* lo que significa que el *stack pointer* siempre apunta al último elemento de la pila y que el puntero se predecrementa cada vez que se realice una nueva inserción (push). Puesto que la pila crece de manera descendente (*downwards*) lo normal es que el puntero a pila o *stack pointer* esté inicializado al límite superior de la SRAM. Así, si la SRAM ocupa el rango

de memoria de 0x200000000 a 0x20007FFF el puntero a pila comenzará en la dirección 0x20008000 de tal forma que la primera operación tipo push que se realice sobre la pila será ubicada en la dirección 0x20007FFC ya que, como indicábamos anteriormente, el puntero a pila está predecrementado. El segundo elemento insertado en la pila se haría en la posición 0x2000FF8 tal y como puede apreciarse en la Figura 16 [tag15].

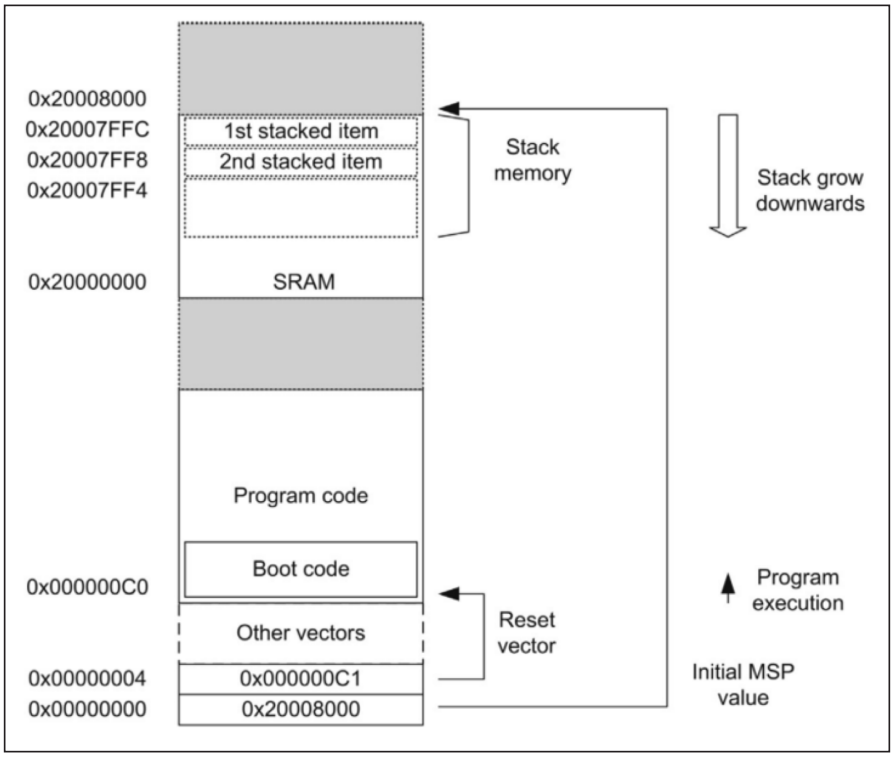

Figura 16. Ejemplo de dos elementos insertados en la pila

6
LENGUAJE MÁQUINA

En el capítulo anterior estudiamos el conjunto de reglas que determinan cómo debe llevarse a cabo la generación de código máquina a partir de código de alto nivel. Durante este capítulo y, estrechamente relacionado con las reglas que el ABI establece, describiremos el conjunto de instrucciones disponibles para el procesador Cortex Mo+. El conjunto de instrucciones que un procesador puede ejecutar recibe el nombre de repertorio de instrucciones o ISA (*Instruction Set Architecture*).

Una cuestión fundamental de un ISA es conocer los diferentes modos de direccionamiento disponibles o, lo que es lo mismo, cómo se pueden obtener los operandos sobre los que las instrucciones se llevarán a cabo. En este capítulo estudiaremos los diferentes modos de direccionamiento existentes y veremos cuáles de ellos están disponibles y cómo se implementan en el ISA del ARM.

Si bien es cierto que el uso de lenguajes de programación como C o C para Arduino nos ofrece mecanismos para acceder a detalles de muy bajo nivel propios de la arquitectura, como puede ser a través del uso de punteros para la gestión de memoria, adquirir unas ciertas nociones básicas de las instrucciones en ensamblador puede llegar a ser muy útil para tareas de optimización de código o depuración. Es este el motivo por el que durante este capítulo exploraremos el repertorio de instrucciones del procesador Cortex Mo+ y algunas otras cuestiones propias del lenguaje máquina.

6.1. EL CONJUNTO DE INSTRUCCIONES DEL CORTEX M0+

La capacidad de un procesador para desarrollar una determinada actividad viene determinada por el conjunto de instrucciones que éste puede ejecutar y cómo éstas se organicen secuencialmente en programas. Cada instrucción del conjunto de instrucciones define una operación más básica que puede ser ejecutada directamente por el procesador que podrá consistir en una operación aritmético-lógica, acceso a memoria, bifurcación, etc.

Aunque como programadores de bajo nivel utilicemos nemotécnicos para referirnos a esas instrucciones básicas, a los que nos referiremos como instrucciones en ensamblador, el procesador sólo podrá ejecutar código binario (secuencia de unos y ceros) al que nos referiremos como código máquina. Una instrucción, generalmente, se ejecutará sobre unos operandos por lo que del conjunto de bits que componen la instrucción, una parte estará dedicada a identificar la instrucción a ejecutar y otra a especificar de dónde obtener los operandos. Internamente el procesador necesita un hardware específico para llevar a cabo la decodificación de la instrucción que consistirá básicamente en determinar la operación a ejecutar y los operandos sobre los que se ejecutará.

La capacidad de un procesador vendrá, por lo tanto, determinada por el conjunto de operaciones que éste ofrezca. En cualquier caso, habrá un mínimo de operaciones para que un procesador sea funcional, como por ejemplo:

- Instrucciones para el procesamiento de datos, de tipo aritméticas como ADD o SUBSTRACT o lógicas como AND u OR.
- Instrucciones de acceso a memoria (lectura o escritura).
- Instrucciones para el control del flujo de programa como las bifurcaciones, bifurcaciones condicionales o las llamadas a procedimientos.

Además de estas instrucciones básicas, para el caso concreto del procesador Cortex M0+ que nos ocupa, podremos disponer de las siguientes instrucciones:

- Manejo de excepciones y soporte al sistema operativo
- Acceso a registros especiales.
- Operaciones tipo *sleep*., que detienen partes del microcontrolador para reducir consumo.
- Barreras de memoria.

El procesador Cortex M0+ soporta un conjunto de 56 instrucciones, algunas con una extensión de 16 bits y otras con extensión de 32 bits. La Figura 17 enu-

mera la lista de operaciones disponibles, organizada en dos tablas, una con las operaciones de 16 bits y otra con las de 32 bits.

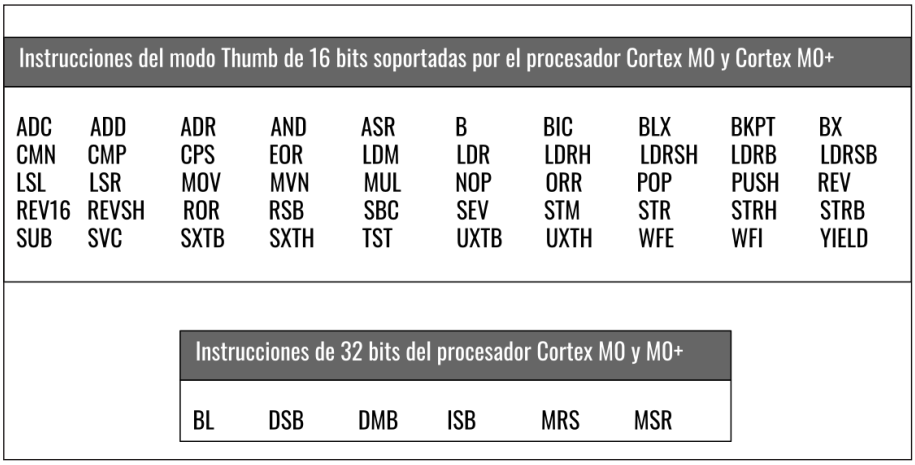

Figura 17. Lista de instrucciones soportada por el procesador Cortex M0+

6.2. LA SINTAXIS DEL LENGUAJE ENSAMBLADOR

Una instrucción en ensamblador utiliza la siguiente sintaxis:

```
Label: nemotécnico operando1, operando2,...; Comentarios
```

La etiqueta *label* se utiliza como una referencia a una parte del código y es opcional. Después, la instrucción especificará el nemotécnico asociado a la operación a ejecutar y el operando u operandos.

En cuanto al orden de los operandos, éste dependerá del tipo de instrucción, así:

- Para instrucciones de procesamiento de datos el primer operando es el destino de la operación.
- Para operaciones de lectura en memoria, el primer operando, generalmente es el registro en el que el dato se cargará.
- Para las operaciones de escritura en memoria, el primer operando generalmente es el registro que almacena el dato a escribir en memoria.

El número de operandos que tendrá una instrucción no es fijo y dependerá de la instrucción en cuestión. Algunas instrucciones puede que incluso no tengan

ningún operando. Además, hay nemotécnicos que se pueden utilizar con diferentes tipos de operandos como por ejemplo la instrucción MOV utilizada para la transferencia de datos entre dos registros o para poner un valor inmediato (una constante) en un registro. El número de operandos en este caso dependerá de lo que se desee realizar:

```
MOVS R0, #0x12 ; Set R0 = 0x12 (hexadecimal)
MOVS R1, #'A'; Set R1 = ASCII character A
```

6.2.1. LA LISTA DE INSTRUCCIONES

A continuación detallaremos el conjunto de instrucciones que puede ejecutar el procesador Cortex Mo+ organizada en grupos dependiendo de su funcionalidad.

6.2.1.1. MOVIMIENTO DE DATOS DENTRO DEL PROCESADOR

La transferencia de datos es una de las tareas más comunes de un procesador, para lo cual, se utilizará el nemotécnico MOV que podrá aceptar diferentes tipos de operandos.

- `MOV <Rd>, <Rm>; Mueve un registro a otro registro.`
- `MOVS <Rd>, <Rm>; Mueve un registro a otro registro y actualiza el APSR o registro de estado al mismo tiempo.`
- `MOVS <Rd>, #immed8; Mueve un dato inmediato (signo extendido) a un registro.`
- `MRS <Rd>, <SpecialReg>; Mueve un registro especial en un registro.`
- `MSR <SpecialReg>, <Rd>; Mueve un registro a un registro especial.`

6.2.1.2. ACCESO A MEMORIA

El procesador Cortex Mo+ soporta una serie de operaciones de acceso a memoria con diferentes modos de direccionamiento y tamaño de transferencia de datos. Los tamaños de transferencia de datos pueden ser de una palabra word, media palabra half word y Byte. Además, también hay instrucciones diferentes para la transferencia de datos con y sin signo.

- `LDR <Rt>, [<Rn>, <Rm>]; Word read: Lee un dato en memoria y lo carga en un registro.`
- `LDR <Rt>, [<Rn>, #immed5]; Word read: Lee un dato en memoria y lo carga en un registro. Utiliza un direccionamiento relativo a un registro base.`
- `LDR <Rt>, [PC, #immed8]; Word read: Lee un dato en memoria y lo carga en un registro. Utiliza un direccionamiento con desplazamiento. Utiliza un direccionamiento con desplazamiento relativo al contador de programa.`
- `LDR <Rt>, [SP, #immed8]; Word read: Lee un dato en memoria y lo carga en un registro. Utiliza un direccionamiento con desplazamiento. Utiliza un direccionamiento con desplazamiento relativo al puntero a pila.`

Por simplificación sólo se han listado las instrucciones de lectura de un dato en memoria del tamaño de palabra. Las mismas instrucciones pueden aplicarse para la media palabra y el byte utilizando el nemotécnico acabo en H o B, como por ejemplo LDRH o LDRB para la lectura sin signo. De la misma manera, tenemos las instrucciones de escritura que utilizarán el mismo formato de instrucción cambiando el nemotécnico que será el STR con los sufijos indicados para la media palabra y el byte.

6.2.1.3. ACCESO A LA PILA

El manejo de la pila se realizará utilizando dos instrucciones: la instrucción PUSH para decrementar el puntero a pila o SP y meter un dato en la pila y la instrucción POP para leer un dato de la pila e incrementar el SP. Ambas instrucciones permite utilizar múltiples registros como operandos.

- `PUSH <Ra>, <Rb> ,..; Escribe uno o varios registros en la pila y actualiza el puntero a pila (decremento).`
- `POP <Ra>, <Rb> ,..; Lee uno o varios registros de la pila y actualiza el puntero a pila (incremento)`

El uso de la pila es fundamental para implementar la llamada a procedimientos y, más específicamente, la capacidad de utilizar el registro LR o el PC con las instrucciones PUSH y POP ya que de esta manera se puede combinar la

restauración de los valores de los registros y la operación de retorno de la llamada a procedimiento en una única instrucción. A continuación se muestra un ejemplo

```
my_function
PUSH R4, R5, R7, LR; Save R4, R5, R7 and LR (return address)
... ; function body
POP R4, R5, R7, PC ; Restore R4, R5, R7 and return
```

La Figura 18 representa el estado de la pila después de haber realizado la operación PUSH con múltiples registros.

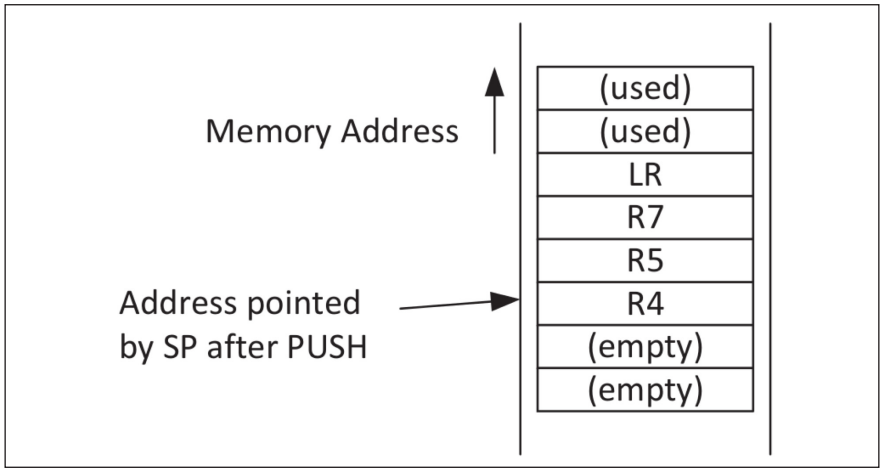

Figura 18. Representación del estado de la pila después de una operación tipo PUSH

6.2.1.4. OPERACIONES ARITMÉTICAS

La mayoría de las operaciones aritméticas se llevarán a cabo entre dos registros o entre un registro y un valor inmediato. A continuación listamos algunas de las operaciones más básicas que puede admitir variaciones como la actualización del registro del estado, utilizar como operando el contador de programa (PC), uso de acarreo, etc.

- `ADDS <Rd>, <Rn>, <Rm>: Suma de dos registros.`
- `ADDS <Rd>, <Rn>, #immed3: Suma un valor inmediato a un registro.`
- `SBCS <Rd>, <Rd>, <Rm>: Resta con acarreo.`
- `MULS <Rd>, <Rm>, <Rd>: Multiplicación.`
- `CMP <Rn>, <Rm>: Comparación.`

6.2.2. OPERACIONES LÓGICAS

- ANDS <Rd>, <Rd>, <Rm>: Y lógico.
- ORRS <Rd>, <Rd>, <Rm>: O lógico.
- EORS <Rd>, <Rd>, <Rm>: O exclusivo lógico.
- BICS <Rd>, <Rd>, <Rm>: Clear a nivel de bit lógico.
- MVNS <Rd>, <Rm>: NOT a nivel de bit.
- TST <Rn>, <Rm>: AND a nivel de bit.

6.2.2.1. OPERACIONES DE ROTACIÓN Y DESPLAZAMIENTO

Las operaciones de desplazamiento de bit son muy útiles cuando se realiza cierto tipo de operaciones.

- ASRS <Rd>, <Rd>, <Rm>: Desplazamiento a nivel de bit a la derecha.
- RORS <Rd>, <Rd>, <Rm>: Rotación a la derecha.

6.2.2.2. CONTROL DE FLUJO DEL PROGRAMA

El Cortex Mo soporta 5 tipos de instrucciones de bifurcación esenciales para el control del flujo de ejecución de un programa, permitiendo así que el código pueda estar organizado en funciones y subrutinas.

- B <label>: Bifurcación incondicional a una dirección.
- B<cond> <label>: Dependiendo del registro de estado bifurca a una dirección.
- BL <label>: Bifurcación a una dirección almacenando previamente la dirección de retorno en el registro LR. Esta instrucción se utiliza normalmente en la llamada a procedimientos.
- BX <Rm>: Bifurcación a una dirección especificada en un registro cambiando el estado del procesador dependiendo del bit 0 del registro. Como el Cortex M0+ sólo soporta el código Thumb el bit 0 deberá estar puesto a 1 porque de otra forma estaría intentando cambiar al estado ARM.

Entre las etiquetas más comunes que se pueden utilizar como condición *cond* destacamos las siguientes:

- EQ: Igual.
- NE: No igual.
- LS: Menor que.
- GE: Mayor que.

7
ENTRADA Y SALIDA

El sistema de entrada/salida constituye, junto con el procesador y la memoria una de las partes esenciales de un computador. En concreto, la entrada/salida será la que permita al computador comunicarse con el mundo exterior y permitir la interacción entre la persona y el computador o el computador y otros dispositivos.

Desde el primer ejemplo que se desarrollará con la placa Arduino, el *"Hola, Mundo"*, ya se introduce cierta interacción con el exterior, ya que es necesario comunicar un mensaje por el puerto serie. Esta versión del *"Hola, Mundo"* difiere un poco del que normalmente se desarrolla cuando abordamos el aprendizaje de un nuevo lenguaje. En este caso, la plataforma utilizada carece de periférico de salida estándar, papel que generalmente es desarrollado por una pantalla en un PC. Para adaptarnos, hemos utilizamos nuestro propio equipo para escuchar lo publicado en el canal serie e imprimirlo en la pantalla de nuestro equipo. Así pues, aunque la pantalla es el ejemplo paradigmático de periférico de salida, también lo es el puerto serie utilizado en nuestro *"Hola, Mundo"*.

En el sistema de entrada/salida los buses juegan un papel fundamental pues son los encargados de establecer el camino por el que viajarán datos, direcciones, instrucciones y señales de control. La Figura 19, extraída de [tag15] representa el esquema de conexionado entre procesador y el resto de los elementos (memorias y periféricos) a través de los buses del sistema para el caso del Cortex Mo+.

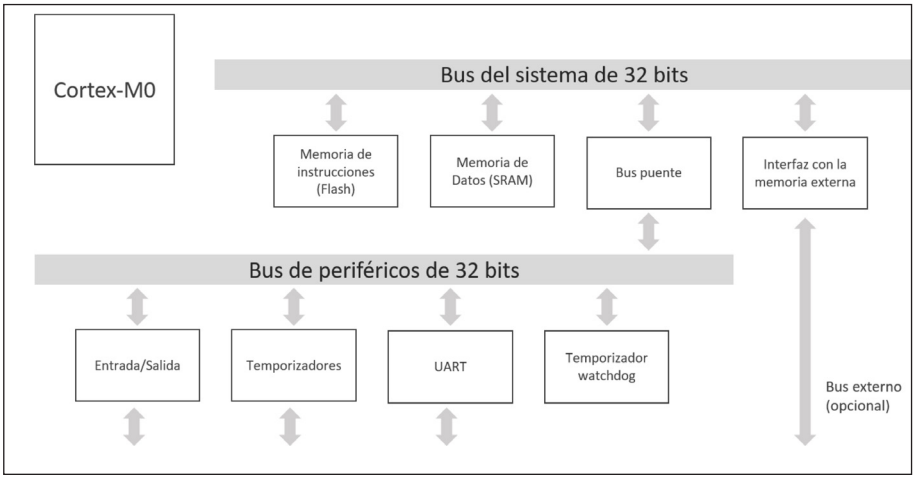

Figura 19. Bus de sistema y periféricos conectando los diferentes elementos de un computador

7.1. TÉCNICAS DE ENTRADA/SALIDA

Una cuestión fundamental de las operaciones de entrada y salida es determinar cómo se llevará a cabo la comunicación entre el periférico y el procesador que, en términos generales, requerirá una primera fase de sincronización entre ambos para proceder, posteriormente, a la transferencia de información. Estas fases de sincronización y transferencia pueden gestionarse de diferente manera, de ahí que podamos hablar de tres tipos de técnicas para la gestión de la comunicación entre periféricos y procesador.

Una primera técnica, la menos eficiente en términos de la sobrecarga que este modo de proceder supone para el procesador, consiste en la entrada/salida por programa o consulta (*polling*). El mecanismo de sincronización requiere, por parte del procesador, el chequeo continuo o periódico para determinar cuándo el periférico está listo para llevar a cabo la transferencia de datos. El proceso de consulta implica que un determinado número de ciclos se invertirán, periódicamente, en chequear el estado del periférico.

Una segunda técnica, algo más eficiente, es la entrada/salida con interrupciones. En este caso, el procesador no necesita chequear constantemente el estado del periférico sino que será éste el que, cuando esté preparado para la transferencia, notifique al procesador mediante una interrupción. Así, mientras el periférico no esté listo, el procesador puede invertir su tiempo de cómputo en otras tareas.

Por último, la entrada/salida con acceso directo a memoria (*Direct Memory Access*, DMA) es la más eficiente por cuanto la transferencia de datos se lleva a cabo directamente entre el periférico y la memoria, sin que el procesador tenga que intervenir en el proceso de transferencia. La única limitación a la transferencia y el rendimiento del procesador está en el uso de los buses del sistema, pues los datos pasan del periférico a la memoria utilizando los buses del sistema que, a su vez, son los que emplea el procesador para la captura de instrucción y búsqueda de operandos en memoria. Esta es la única limitación al rendimiento de la transferencia implementado esta técnica.

Los periféricos de un microcontrolador como el que utiliza Arduino Zero se controlan a través de interfaces como entradas y salidas analógicas y digitales, UARTs, I2C, SPI o USB. En el microcontrolador Cortex Mo+ de Arduino Zero, los periféricos se controlan a través de registros mapeados en memoria de tal forma que la interacción con un periférico se traducirá en operaciones de lectura y escritura sobre determinados registros (o posiciones de memoria). Es importante hacer hincapié en el hecho de que los registros de un periférico no tienen nada que ver con los registros del procesador. Es más, los registros asociados a un periférico están mapeados en memoria lo que supone que la lectura y escritura se realizará utilizando instrucciones que accedan a la memoria para leer o escribir.

8
SESIONES DE PRÁCTICAS

En las próximas secciones se describen cada una de las sesiones de laboratorio a desarrollar en la asignatura de Estructura de Computadores. Estas sesiones se han estructurado de modo que se alternan sesiones con el contenido práctico que hace referencia a los diferentes temas de la asignatura con sesiones de evaluación sobre cada contenido.

9
SESIÓN 1. EXPERIMENTANDO CON LA ARQUITECTURA HARVARD DE ARDUINO ZERO

Esta sesión de laboratorio servirá como toma de contacto con el hardware y software que se utilizará para las siguientes sesiones de laboratorio. Esta sesión estará guiada por el desarrollo de una serie de ejercicios.

9.1. "HOLA, MUNDO"

Como cada vez que nos enfrentamos a un nuevo lenguaje de programación o plataforma, el primer ejercicio a realizar es el *"Hola, Mundo"*. Este ejercicio consiste en imprimir esta cadena de texto por la salida estándar. En este caso no tenemos ningún terminal de salida, haremos un pequeño ajuste para que el mensaje se envíe al puerto serie. Utilizaremos el monitor de puerto serie que viene con el IDE de Arduino para recoger lo transmitido a través de ese puerto.

Si ya hemos seguido los pasos previos de configuración del entorno, tal y como se han descrito en los anexos previos, sólo tendremos que asegurarnos de que el puerto seleccionado es el apropiado. Para ello, en el menú *Herramientas → Puerto: ...* → confirmaremos que está seleccionado el puerto en cuestión, tal y como puede apreciarse en la Figura 20.

Una vez confirmado el puerto, el siguiente paso será escribir el código, para lo cual, en el menú *Archivo → Nuevo* abriremos un nuevo proyecto que ya viene con los esqueletos de las funciones *setup* y *loop*. En este primer programa, sólo imprimiremos la cadena de texto al inicio del programa por lo que la función *loop* quedará vacía.

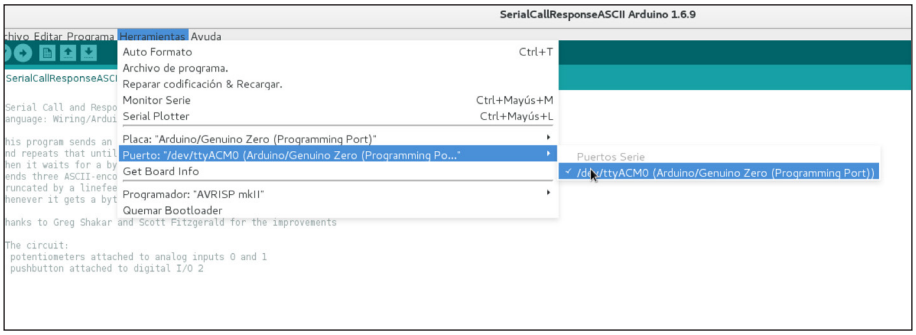

Figura 20. Selección del puerto Serie

```
void setup() {
// put your setup code here, to run once:
Serial.begin(9600);
Serial.print("Hello, World");
}
void loop() {
// put your main code here, to run repeatedly:
}
```

El objeto *Serial* será el utilizado para la configuración e interacción con el puerto serie. En primer lugar, tendremos que indicarle a la velocidad a la que nos comunicaremos con el puerto serie, es decir la cantidad de bits por segundo a la que se transfiere la información a través de este puerto. Como podemos observar en la línea 3, hemos establecido esa velocidad a 9600 bits/s.

El objeto *Serial* tiene el método *print* para el envío de datos al puerto serie. En este caso, se envía la cadena de texto *"Hello, World"*.

Figura 21. Monitor serie del IDE de Arduino

Tras verificar, compilar y subir el programa a nuestra placa Arduino Zero (ver Anexo A), será necesario conectarse al puerto serie para observar la información que se envía por él. Para ello, utilizaremos el monitor que nos ofrece el IDE en el menú *Herramientas → Monitor Serie*. Como se puede observar en la Ilustración 16.

La placa está ahora programada para imprimir el mensaje una sola vez, por ello, si no teníamos abierta la ventana del monitor serie cuando cargamos el programa por primera vez, lo que podemos hacer es pulsar el botón de reseteo de la placa y veremos como aparece el mensaje, como se observa en la Figura 22.

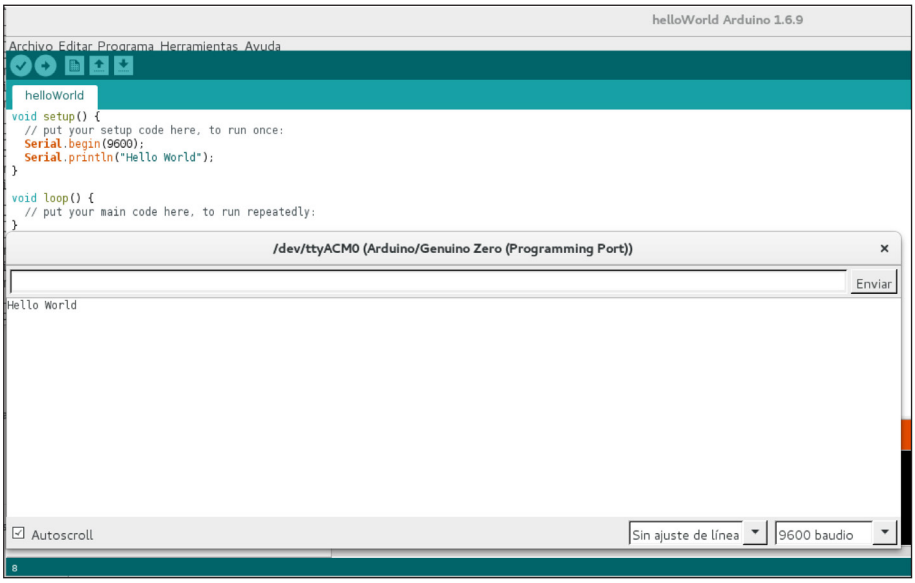

Figura 22. Cadena recogida del puerto serie por el monitor serie

9.2. EXPLORANDO LA MEMORIA

El tema de memoria será tratado, con más detalle, durante la Sesión 3. Sin embargo, durante esta sesión analizaremos el mapa de memoria del Arduino Zero y, más concretamente, los usos que los diferentes tipos de memoria tienen (datos o instrucciones).

Para entender esta práctica, bastará con saber que los diferentes tipos de memoria que tiene el Arduino Zero estarán mapeadas a las siguientes direcciones de memoria:

Tabla 5. Mapa de memoria del Arduino Zero.

Memoria	Dirección de inicio
Flash interna	0x00000000
SRAM interna	0x20000000
Bridge periférico A	0x40000000

El propósito de este ejercicio es comprobar que la plataforma Arduino Zero implementa una arquitectura Harvard modificada para lo cual haremos una serie de ejercicios para explorar las diferentes direcciones físicas en las que se localizan los datos e instrucciones de un programa.

Partiendo del mapa de memoria detallado en la Tabla 5, el siguiente ejercicio pretende demostrar que, efectivamente, los datos e instrucciones están en memorias separadas.

Sabemos que las instrucciones del programa se cargan en la memoria Flash interna por lo que bastará con demostrar que las instrucciones de nuestro programa están en direcciones de memoria que comiencen por oxo... tal como indica la Tabla 5. Por su parte, los datos se almacenan en la SRAM interna que, como también sabemos, se trata de una memoria volátil. Así, declararemos dos tipos de variables: una variable estática (variable cad) inicializada y una variable local o automática inicializada (variable a), como se muestra en el siguiente código:

```
    Static char cad[]="Hola, Mundo";
    void setup(){
    char text[32];
    Serial.begin(9600);
    int a =1;
sprintf(text, "Direccion de cad: %x\nDireccion de a: %x\n", &a, &cad);
    Serial.print("Hello, World");
    }
    void loop(){

    }
```

El resultado de ejecutar este código será la impresión de la dirección de memoria donde las variables cad y a están almacenadas, como se muestra en la Figura 23.

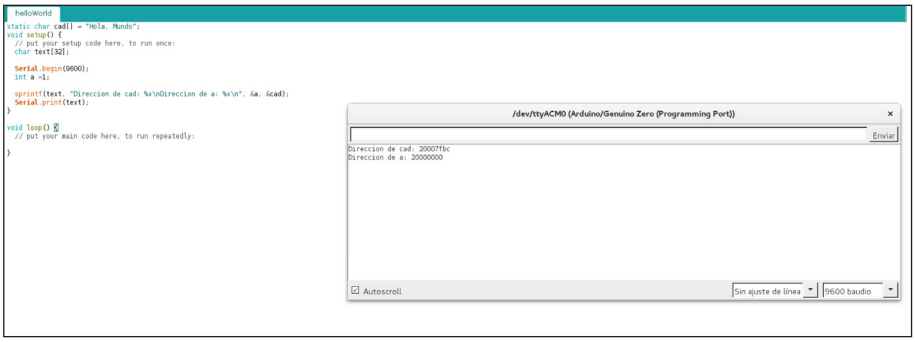

Figura 23. Impresión de las direcciones de memoria de dos variables

Para comprobar que las instrucciones están almacenadas en la memoria Flash interna, utilizaremos el depurador (ver Anexo B) para explorar la memoria, así como el código ensamblador generado.

Para lanzar openOCD para la depuración remota, puede ser necesario tener que desconectar y volver a conectar la placa si el puerto estuviera bloqueado por el IDE de Arduino.

Sabremos si esto es necesario cuando al lanzar el openOCD obtenemos el siguiente error:

```
$ openocd -f /usr/share/openocd/scripts/target/arduino_zero.cfg
Open On-Chip Debugger 0.9.0 (2015-05-28-17:08)
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'swd'
adapter speed: 500 kHz
adapter_nsrst_delay: 100
cortex_m reset_config sysresetreq
Error: unable to find CMSIS-DAP device
```

Si este fuera el caso, la solución consistirá en desconectar la placa del conector USB y volver a conectarlo. Por lo tanto, el primer paso será poner en marcha el OpenOCD:

```
$ openocd -f /usr/share/openocd/scripts/target/arduino_zero.cfg
Open On-Chip Debugger 0.9.0 (2015-05-28-17:08)
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'swd'
adapter speed: 500 kHz
adapter_nsrst_delay: 100
cortex_m reset_config sysresetreq
Info : CMSIS-DAP: SWD Supported
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : CMSIS-DAP: FW Version = 02.01.0157
Info : SWCLK/TCK = 1 SWDIO/TMS = 1 TDI = 1 TDO = 1 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 500 kHz
Info : SWD IDCODE 0x0bc11477
Info : at91samd21g18.cpu: hardware has 4 breakpoints, 2 watchpoints
```

Seguidamente lanzamos el depurador KDbg con las siguientes opciones y seleccionamos el ejecutable como se observa en la Figura 24:

```
$ kdbg -r :3333
```

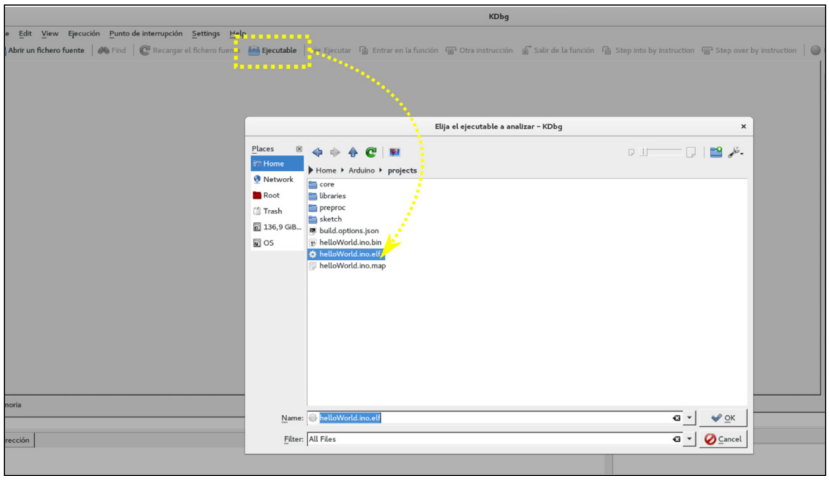

Figura 24. Selección del ejecutable desde el menú del depurador KDbg

Después de seleccionar el ejecutable aún no nos aparecerá ningún código en la ventana principal de KDbg, para ello, tendremos que indicarle dónde está ubicado

el código fuente 1 que, en nuestro caso estará en el directorio *Arduino/helloWorld*. Es importante recordar que, por defecto, el cuadro de diálogo sólo mostrará aquellos archivos con extensión .c o .cpp por lo que habrá que borrar el filtro para que muestre todos los archivos de código. Para ello, bastará con pulsar el icono que aparece al final de cuadro de texto Filter, tal y como se aprecia en la Figura 25.

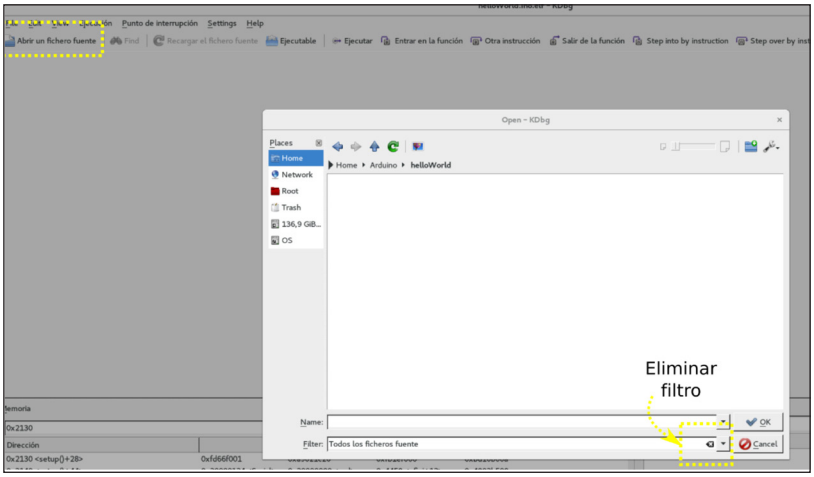

Figura 25. Eliminando el filtro del menú de selección del código fuente de KDbg

Una vez eliminado el filtro aparece el fichero con extensión .ino como se observa en la imagen Figura 26.

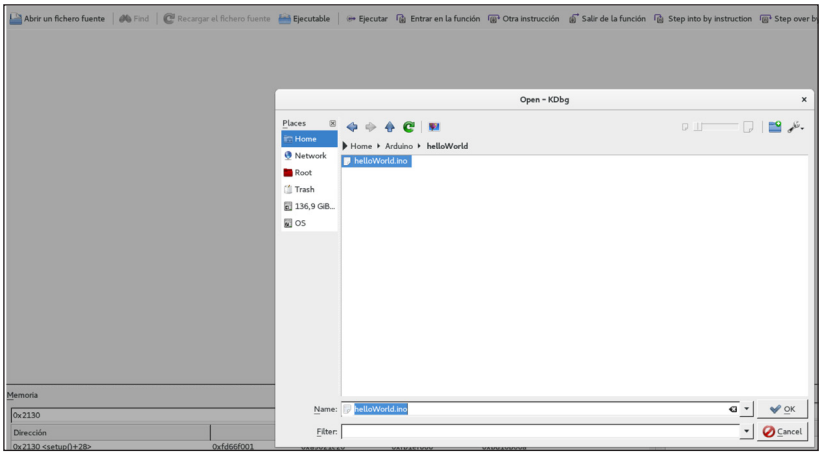

Figura 26. Selección del código fuente una vez eliminado el filtro

Una vez cargado el ejecutable y el código fuente en el KDbg ya podremos explorar el contenido de la memoria. Así, si queremos conocer el contenido de las direcciones de memoria que hemos impreso anteriormente por el puerto serie (0x20007fbc y 0x20000000) bastará con introducir dichas posiciones en la ventana del KDbg que explora el contenido de la memoria. El contenido de la memoria se está representando como caracteres y así podemos observar como para cada posición de memoria (por cada byte) se muestra el valor ASCII y el carácter correspondiente, pudiéndose leer la cadena "Hola, mundo".

El objetivo de este ejercicio era demostrar que los datos e instrucciones de un programa se almacenaban en memorias diferentes. Habiendo comprobado entonces que los datos están almacenados en posiciones de memoria que comienzan por 0x2… correspondientes a la memoria SRAM interna, pasaremos a demostrar que las instrucciones se encuentran en la Flash interna. Para ello, podemos comenzar por desplegar el menú asociado a cada una de las instrucciones del programa, pulsando el símbolo + que encontramos justo a la izquierda del número de línea. Al desplegar la instrucción podemos observar el código ensamblador asociado a la instrucción del código de alto nivel. Por ejemplo, la Figura 27 muestra el código ensamblador generado a partir de la instrucción de inicialización del puerto serie *Serial.begin(9600)*.

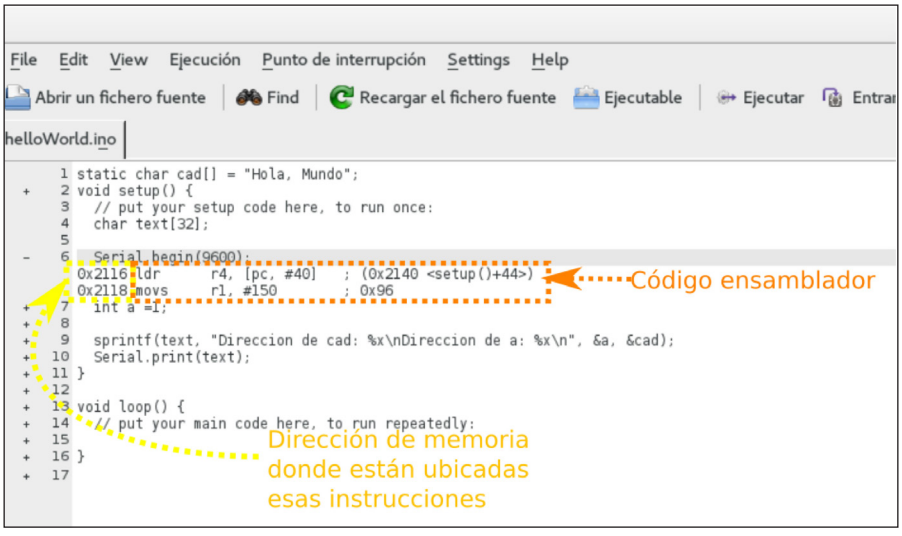

Figura 27. Explorando la memoria y el código ensamblador

Como puede observarse en la Figura 27, las instrucciones seleccionadas están almacenadas en las posiciones de memoria 0x2116 y 0x2118. Realmente estas direcciones son la dirección 0x00002116 y la 0x00002118. Como era de esperar, las instrucciones o lo que es lo mismo, el código a ejecutar por el procesador está almacenado en la memoria Flash interna que a su vez está mapeada a direcciones de memoria que comienzan por 0x0… como es el caso.

Sin embargo, si queremos comprobar que esto es del todo cierto, aprovechando que tenemos el ejecutable cargado en el KDbg podemos explorar la memoria utilizando la ventana Memoria como se observa en la Figura 28.

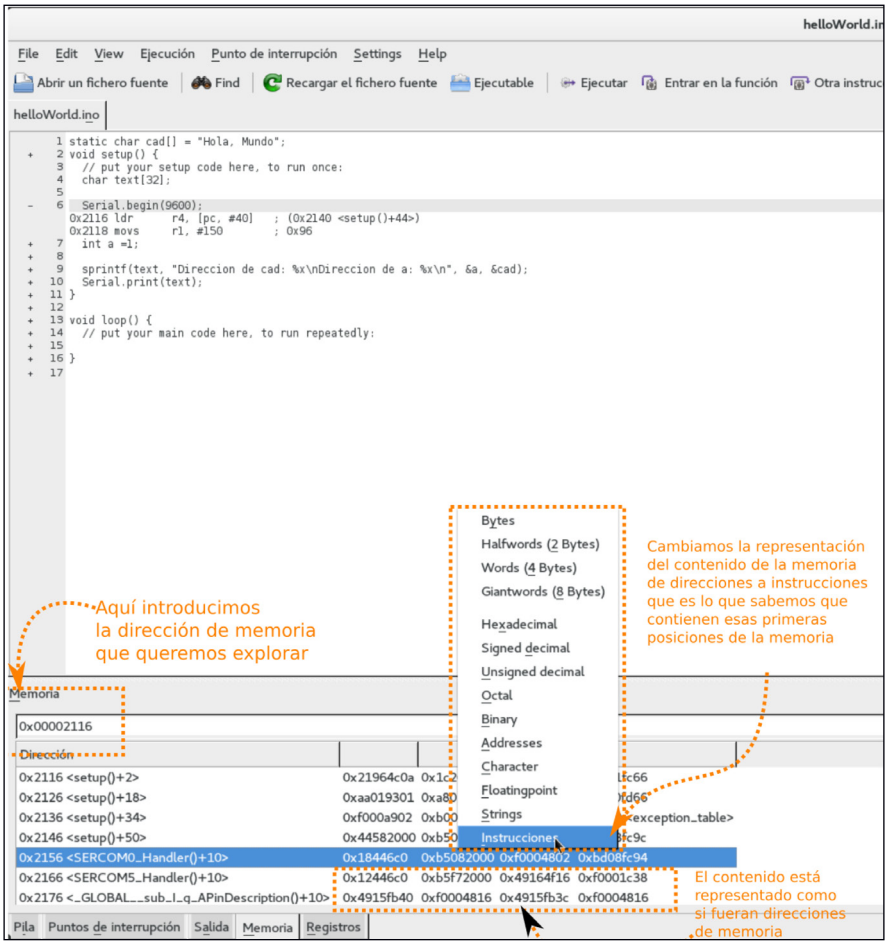

Figura 28. Ajustando el formato del contenido de la memoria en una determinada posición

10
SESIÓN 2. EVALUACIÓN DE LA ARQUITECTURA HARVARD

10.1. SESIÓN 2: PREGUNTAS

Responde a las cuestiones planteadas sobre el siguiente fragmento de código:

```
void setup() {
Serial.begin(9600);

}
void loop() {
Serial.print("Hello, World");
delay(1000);

}
```

- **Pregunta 1.** Determina si la cadena de texto "Hola, Mundo" es considerada como parte de las instrucciones del programa o los datos. Explica el proceso seguido.
- **Pregunta 2**. ¿Hay algún dato almacenado en la memoria de datos?
- **Pregunta 3.** ¿Qué proporción de la memoria de instrucciones está siendo ocupada por el programa? ¿Y de la de datos?
- **Pregunta 4.** En términos de uso de recursos, como la memoria, compara el programa dado con la siguiente versión, indicando cuál de los dos es más eficiente o si los dos emplean los mismos recursos:

```
Static const char cad[]=”Hola, Mundo”;
void setup() {
Serial.begin(9600);

}
void loop() {
Serial.print(“Hello, World”);
delay(1000);

}
```

- **Pregunta 5.** ¿Existe algún mecanismo que nos permita mover contenido de la memoria de datos a la memoria de programa?

10.2. SESIÓN 2: RESPUESTAS

10.2.1. PREGUNTA 1

Para saber en qué memoria está localizada la cadena tendremos que determinar su dirección de memoria, sabiendo que las direcciones de memoria que comienzan por oxo… están mapeadas a la memoria de programa o Flash interna y las que comienzan por ox2… a la memoria de datos o SRAM interna.

Para determinar entonces dicha dirección de memoria recurriremos al uso del depurador EDBG y el KDbg para explorar tanto el contenido de las diferentes memorias y registros como el código ensamblador. Comenzaremos por desplegar el código ensamblador asociado a la sentencia de impresión *Serial.print(“Hola, Mundo”)*. Esta sentencia se compone de tres instrucciones en ensamblador:

```
ldr r1, [pc, #16] ; (0x2124 <loop()+20>)
ldr r0, [pc, #16] ; (0x2128 <loop()+24>)
bl 0x276c <Print::print(char const*)>
```

La instrucción *bl* es la utilizada para saltar a la función *print* encargada de la impresión. Nos centraremos, por lo tanto, en el análisis de las dos primeras instrucciones en las que se realiza el paso de argumentos. La instrucción *ldr* se trata de una instrucción para la carga de una dirección de memoria en un registro.

Aunque esto se estudiará con más detenimiento durante el capítulo 6, cuando estudiemos el lenguaje máquina y los modos de direccionamiento, bastará ahora con que entendamos que lo que estamos cargando en los registros r1 y ro son las direcciones de memoria contenidas en ox2124 y ox2128 respecti-

vamente. Es muy probable que en una de estas dos direcciones de memoria encontremos la cadena que se pasa como argumento a la función *print*, encargada de su impresión.

Así, utilizaremos el KDbg para explorar el contenido de la memoria en esas dos direcciones. La Figura 29 y Figura 30 muestran el contenido de ambas direcciones, donde el formato de representación utilizado es el de Word (4 bytes).

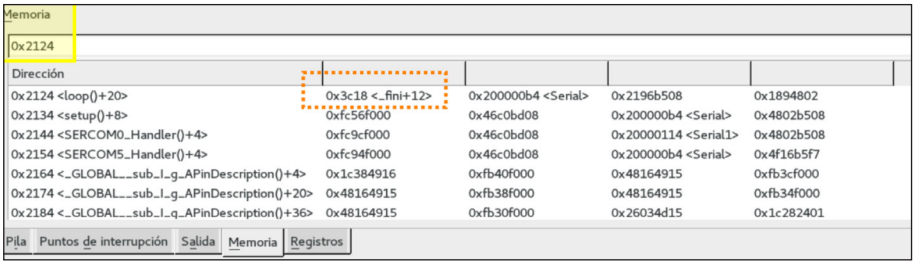

Figura 29. Contenido de la posición de memoria 0x2124

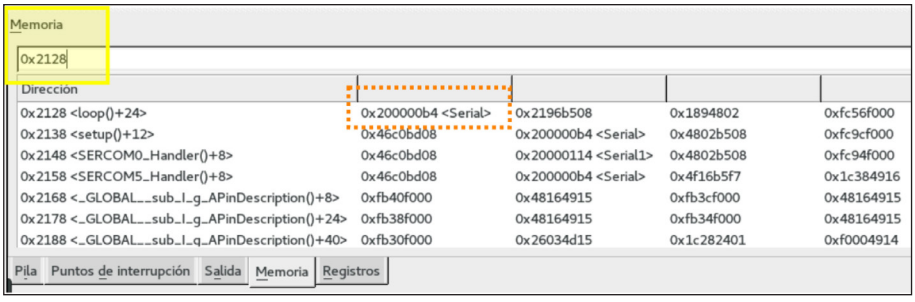

Figura 30. Contenido de la posición de memoria 0x2128

El contenido de la posición 0x2124 se corresponde con una posición de memoria de la memoria de programa o Flash interna. Mientras que la posición 0x2128 almacena una dirección de memoria que se corresponde con la memoria de datos o SRAM. Tendremos que explorar el contenido de ambas posiciones para determinar en cuál de ellas está almacenada la cadena de texto.

En el contenido de la posición 0x3c18 observamos, tal y como se muestra en la Figura 31, que la cadena de texto "Hola, Mundo" está almacenada en la memoria de programa.

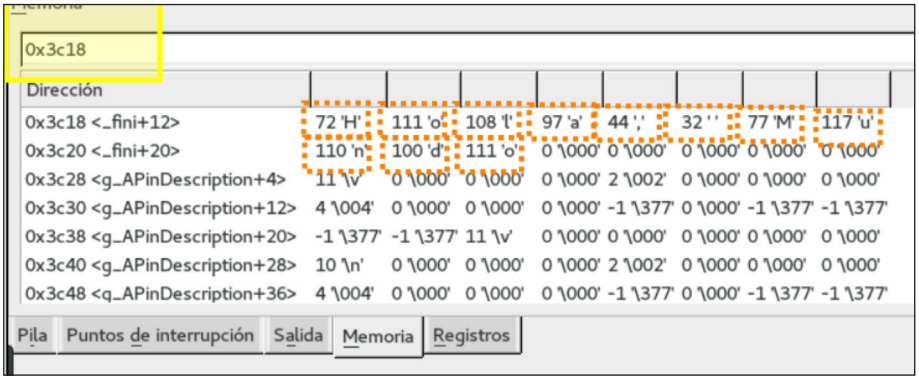

Figura 31. Contenido de la posición de memoria 0x3c18.

10.2.2. PREGUNTA 2

De la respuesta a la pregunta anterior sabemos que al menos, el dato contenido en la posición 0x200000b4 está almacenado en la memoria de datos (SRAM). Sabemos que cada puerto serie tiene asociado un único *Serial Object* que, para el caso del puerto serie 0[10], es referido en código como *Serial*. Este objeto es el que hemos utilizado para imprimir una cadena de texto que será recibida en el otro extremo del canal por la consola monitor serie. En cualquier caso y sin entrar en detalles de cómo se estructuran las instancias de una clase en memoria, bastará con decir que el objeto *Serial* está en la memoria de datos.

10.2.3. PREGUNTA 3

La memoria es un recurso muy limitado en dispositivos empotrados como es el caso de la plataforma Arduino Zero. Por lo tanto, es necesario hacer un uso muy eficiente de la misma para lo cual resulta indispensable conocer cómo el compilador gestiona los diferentes elementos de un programa y en qué secciones de la memoria del programa en ejecución se almacenarán. Aunque todos estos detalles se explorarán a lo largo de las siguientes sesiones, ahora describiremos cómo conocer el tamaño que nuestro programa ocupa en memoria, tanto en la de datos como en la de programa.

Por un lado, el propio IDE de Arduino tras la compilación nos indica el tamaño (en bytes y el tanto por ciento de memoria empleada) del programa compilado (ver Figura 32) pero no indica nada de la memoria de datos (SRAM) empleada. Para ello, la toolchain de AVR nos proporciona una serie de herramientas, entre

10 Pin RX 30 y pin TX 30.

ellas el avr-size. Hay que tener en cuenta que esta herramienta sólo nos informa del consumo de memoria estático, es decir, no tiene en cuenta el consumo de memoria que pueda llevarse a cabo de manera dinámica durante la ejecución del programa.

Es necesario hacer un breve inciso para comprender el resultado que la herramienta avr-size nos devolverá tras la consulta, ya que el tamaño ocupado por nuestro programa, en la SRAM, se devolverá en bytes ocupados en diferentes secciones. Es necesario, por lo tanto, explicar qué se entiende por sección de memoria[11]. Sin entrar en mucho detalle comentaremos que un programa en ejecución se divide en diferentes secciones de memoria. Así, por ejemplo, tenemos la sección .text y .data para las instrucciones y variables (en este caso inicializadas) respectivamente, pero también tenemos otras como la .bss para variables no inicializadas. El compilador es el encargado de determinar qué parte del programa va en cada sección de memoria. Ocurre, sin embargo, que el compilador GCC no interpreta muy bien el concepto de arquitectura Harvard, que implica que datos e instrucciones deben ir a memorias físicas separadas [Whe11].

```
    $ ~/arduino-1.6.9/hardware/tools/avr/bin/avr-size ~/Arduino/projects/hello-
World.ino.elf
    text data bss dec hex filename
    8404 152 1992 10548 2934 /home/mjsantof/Arduino/projects/helloWor
```

Si utilizamos la opción -A y -x obtendremos además información de la dirección a partir de la cual dichas secciones están mapeadas en memoria[12]:

11 En el capítulo 5 estudiaremos el sistema de memoria y, en especial, las secciones de memoria de un programa en ejecución.

12 La ruta ~/arduino-1.6.9/es orientativa y podrá cambiar conforme aparezcan nuevas versiones.

```
$ ~/arduino-1.6.9/hardware/tools/avr/bin/avr-size
~/Arduino/projects/helloWorld.ino.elf
```

section	size	addr
.text	0x20d4	0x2000
.data	0x98	0x20000000
.bss	0x7c8	0x20000098
.ARM.attributes	0x28	0x0
.comment	0x80	0x0
.debug_info	0x277db	0x0
.debug_abbrev	0x389d	0x0
.debug_aranges	0x820	0x0
.debug_ranges	0xf00	0x0
.debug_line	0x5024	0x0
.debug_str	0x8051	0x0
.debug_frame	0x13bc	0x0
.debug_loc	0x525a	0x0
Total	0x427ff	

La sección .bss, como se puede observar, también se encuentra en la memoria SRAM[13]. En este caso aunque nosotros no hayamos declarado expresamente variables no inicializadas, el objeto *Serial* sí que cuenta con miembros de su clase que no están inicializados a ningún valor y que son los que están ubicados en esa sección de memoria.

Por lo tanto, r6espondiendo a la pregunta formulada, la Figura 32 muestra la captura del IDE de Arduino tras la compilación, donde puede verse el porcentaje de memoria sobre el total disponible que ocupa el programa compilado, así como su tamaño en bytes.

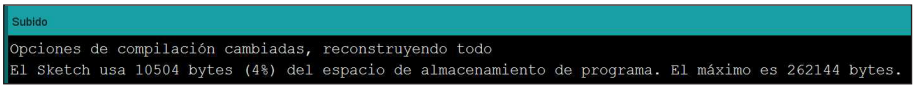

Figura 32. Porcentaje de la memoria de programa utilizada en *"Hola, Mundo"*

13 La memoria SRAM está mapeada a direcciones de memoria que comienzan por 0x2....

10.2.4. PREGUNTA 4

En primer lugar realizaremos la comparación en términos del tamaño que ambos programas ocupan en memoria, para lo que utilizaremos las herramientas presentadas en la respuesta anterior.

La siguiente tabla muestra los valores para las dos versiones del programa *"Hello, World"*, donde la versión 1 es la proporcionada al comienzo de esta sesión y la versión 2 es la que se plantea para esta pregunta. Como se puede observar los valores obtenidos son idénticos.

Tabla 6. Comparación entre tamaño en memoria de las dos versiones del *"Hello, World"*

Parámetros	Versión 1	Versión 2
Tamaño del sketch	8404 bytes	8404 bytes
Tamaño sección data	152 bytes	152 bytes
Tamaño sección .bss	1992 bytes	1992 bytes

Antes de explicar el motivo de que, pese a que el código del programa sea ligeramente diferente, el binario generado parece ser idéntico, haremos una modificación más en la versión 3, tal y como se lista a continuación.

```
char cad[] = "Hola, Mundo";
void setup() {
Serial.begin(9600);
}
void loop() {
Serial.print(cad);
delay(1000);
}
```

La siguiente tabla recoge los valores obtenidos, en términos de bytes en memoria, para la nueva versión del *"Hello, World"*. Como puede apreciarse, hay un ligero descenso del número de bytes que ocupa el programa en cuanto a número de instrucciones así como en el tamaño de la sección de memoria dedicada a las variables inicializadas.

Tabla 7. Resultados de la versión 3 de *"Hello, World"*

Parámetros	Versión 3
Tamaño del sketch	8388 bytes
Tamaño sección data	164 bytes
Tamaño sección .bss	1992 bytes

Para analizar por qué entre la versión 1 y 2 del código no hay ninguna diferencia y con respecto a la versión 3 sí que la hay, debemos centrarnos en cómo el compilador gestiona la cadena de texto en cada una de las versiones, teniendo en cuenta lo siguiente:

- En la versión 1 el texto forma parte de la instrucción.
- En la versión 2 el texto se ha declarado como una variable estática y constante inicializada.
- En la versión 3 el texto se ha declarado como una variable global inicializada.

Hay que tener en cuenta que otras plataformas Arduino, para la versión de código 1 y 2 utilizará memoria tanto en la flash como en la SRAM y era necesario utilizar una versión optimizada de la función *print* para que la cadena de texto no fuera copiada en la SRAM, sino únicamente en la Flash. Sin embargo, en Arduino Zero (Atmel SAMD21G18) esto ya no es así[14] y si utilizamos el depurador para analizar las instrucciones que ejecutan para llevar a cabo la impresión de una cadena se puede comprobar que la misma sólo está almacenada en la memoria Flash y que para la impresión se recurre a una variable *buffer* gestionada dinámicamente, por lo tanto, haciendo más eficiente el consumo de RAM frente a versiones anteriores que hacían una doble reserva de espacio para una misma cadena.

Este es el motivo por el que la versión 1 y 2, en esta placa en concreto, no muestran ninguna diferencia en cuanto al código generado. No sería así en otras plataformas.

Con respecto a la versión 3, se puede observar como la sección .data ha aumentado en 12 bytes que son los que ocupa la cadena. Por otro lado, el tamaño del

14 El Cortex M0+ y otros chips de la familia ARM ya utilizan el mismo espacio de direcciones para SRAM y Flash, y por lo tanto no es necesario que los datos, para ser referenciados, tengan que ser copiados previamente a la SRAM [Sho16].

sketch ha disminuido por cuanto la cadena está ahora contenida en la sección .data y no en la .text.

10.2.5. PREGUNTA 5

Aunque como se ha podido comprobar en la pregunta anterior para el manejo de cadenas no es necesario utilizar ese mecanismo, pues se hace automáticamente, pueden darse otras situaciones donde sea deseable mover contenido de la SRAM a la Flash, teniendo en cuenta que en esta última los datos serán de solo lectura y no podrán ser modificados.

La palabra reservada PROGMEN es un modificador de variables que, según se indica en la web de referencia de Arduino[15] puede utilizarse en distintas posiciones en la sentencia de declaración:

```
const dataType variableName[] PROGMEM = {}; // use this form
const PROGMEM dataType variableName[] = {}; // or this form
const dataType PROGMEM variableName[] = {}; // not this one
```

En la misma web de referencia se ofrecen los siguientes ejemplos de uso concretos:

```
[...]
// save some unsigned ints
const PROGMEM uint16_t charSet[] = { 65000, 32796, 16843, 10, 11234};

// save some chars
const char signMessage[] PROGMEM = {"I AM PREDATOR, UNSEEN COM-
BATANT. ←-
CREATED BY THE UNITED STATES DEPART"};
[...]
```

Como se puede observar, el uso de este modificador está especialmente indicado para forzar que las cadenas de textos no se copien en la SRAM, sino en la memoria Flash, que es un recurso que generalmente está algo menos limitado que la SRAM. Como comentábamos en el ejercicio anterior, si bien es necesario tener en cuenta esta cuestión cuando se trabaje con otras plataformas de Arduino (Uno, Due, etc.) esta cuestión es irrelevante para la plataforma que se utilizará en este laboratorio.

15 https://www.arduino.cc/en/Reference/PROGMEM

11
SESIÓN 3: MEMORIA

Durante esta sesión de prácticas utilizaremos el depurador KDbg para demostrar que, efectivamente, el compilador cumple las reglas establecidas por el ABI en cuestiones que afectan al uso de memoria, como son el alineamiento de datos y la llamada a procedimientos.

11.1. LA PILA

El *layout* o esquema de memoria para el procesador Cortex Mo+ del Arduino Zero, establece que la pila crece de manera descendente a partir del límite superior de la memoria SRAM, ubicado el mismo en la dirección de memoria 0x20007FFF. El *layout* del microcontrolador se representaba en la Figura 9. Este ejercicio consistirá en demostrar que esto es efectivamente así, y que el puntero de pila crece a partir del límite superior de la SRAM hacia posiciones de memoria descendentes. Para ello, utilizaremos el depurador KDbg y su herramienta de exploración de los registros del procesador y la memoria.

El código del programa para analizar el puntero a pila no es relevante puesto lo que queremos ver es a qué dirección de memoria está inicializado el puntero a pila SP y hacia qué dirección crece. Para ello utilizaremos un sketch vacío:

```
void setup(){

}
void loop(){

}
```

Una vez compilado y cargado en la placa, nos interesa analizar el estado de los registros del procesador, especialmente el del registro SP, antes de que el programa comience a ejecutarse. Utilizaremos KDbg para lo cual habrá que lanzar previamente openOCD:

```
    $ openocd -f /usr/share/openocd/scripts/target/arduino_zero.cfg
Open On-Chip Debugger 0.9.0 (2015-05-28-17:08)
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'swd'
adapter speed: 500 kHz
adapter_nsrst_delay: 100
cortex_m reset_config sysresetreq
Info : CMSIS-DAP: SWD Supported
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : CMSIS-DAP: FW Version = 02.01.0157
Info : SWCLK/TCK = 1 SWDIO/TMS = 1 TDI = 1 TDO = 1 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 500 kHz
Info : SWD IDCODE 0x0bc11477
Info : at91samd21g18.cpu: hardware has 4 breakpoints, 2 watchpoints
```

Después lanzamos KDbg, que previamente habremos configurado tal y como se describió en el anexo B:

```
    $ kdbg -r :3333
```

Tras cargar el ejecutable y sin que previamente hayamos comenzado la ejecución del programa a depurar, podemos observar ya el estado inicial de los registros tal y como se muestra en la captura de la Figura 33.

Como se puede observar, el registro SP está inicializado a la dirección 0x20007FFC, siendo la dirección de memoria en la que se escribirá el primer valor cuando se realice una instrucción que introduzca un dato en la pila. Para demostrar ahora que la pila crece hacia posiciones de memoria descendente basta con poner un punto de ruptura justo antes de la llamada a la función *loop()* de tal forma que podamos ver cómo el puntero de pila se decrementa como consecuencia de la llamada a la función.

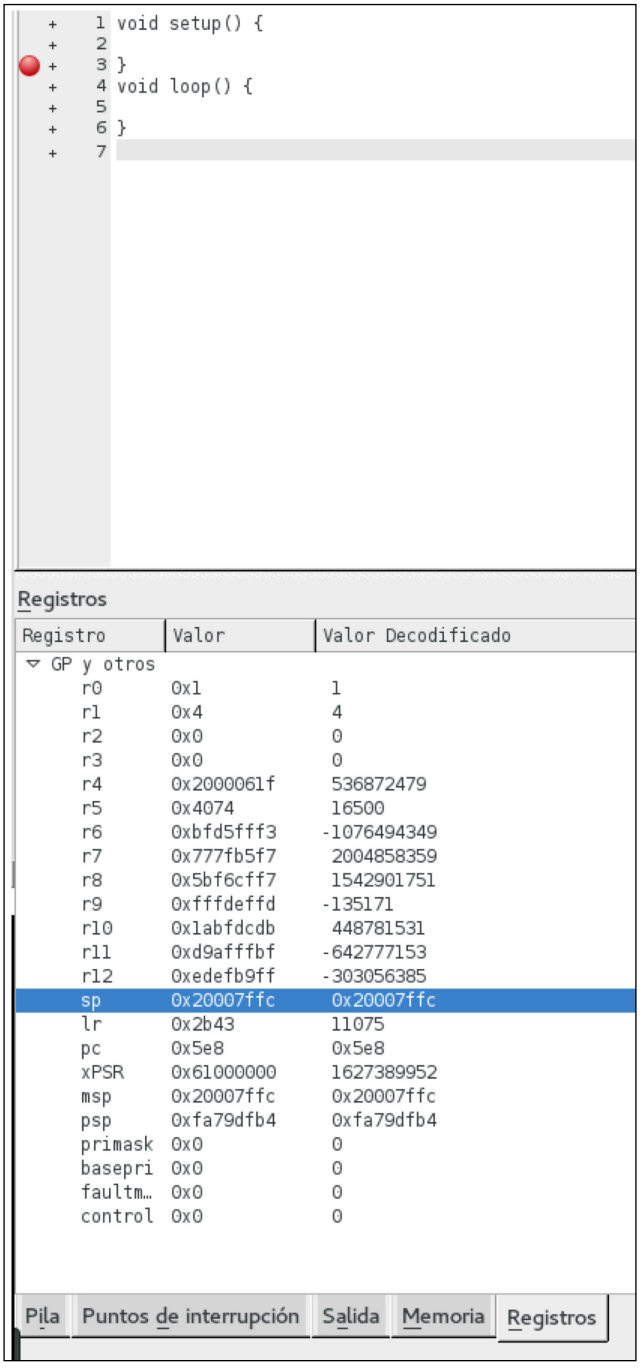

Figura 33. Estado de los registros antes del comienzo de la ejecución del programa

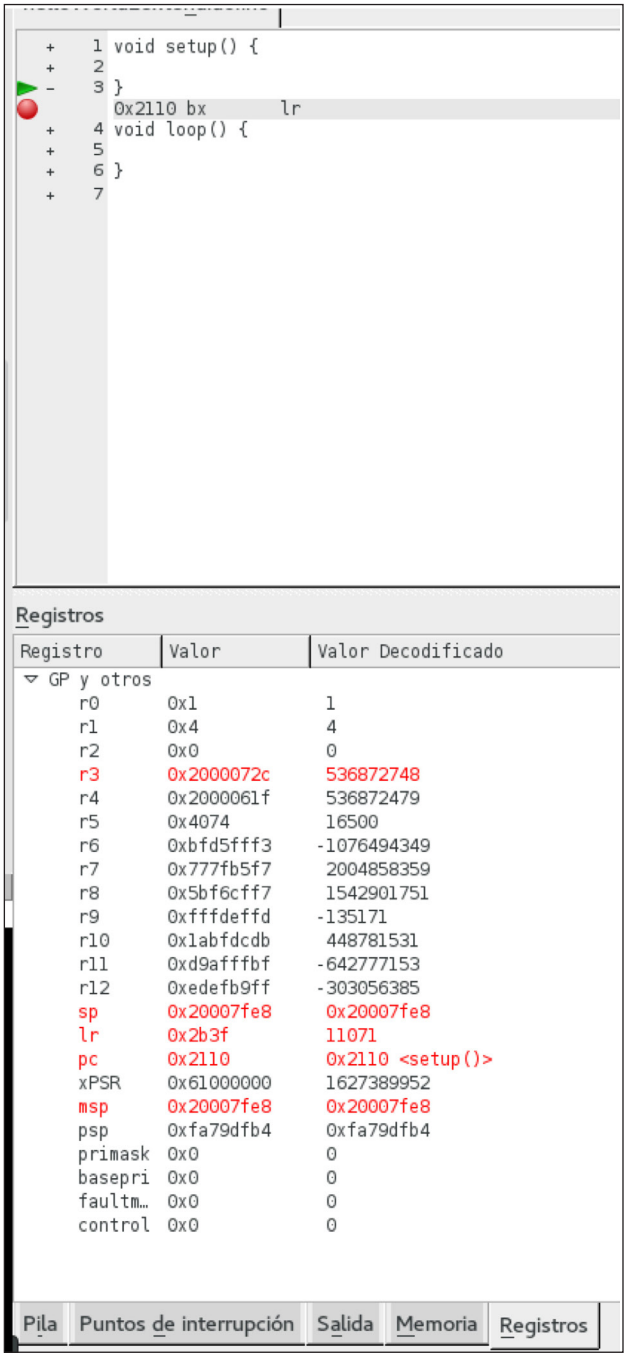

Figura 34. Estado de los registros después de la llamada al procedimiento loop()

Como se observa en la Figura 34, el registro SP pasa a apuntar a la dirección de memoria 0x20007FE8 una vez se realiza la llamada a la función *loop()* como consecuencia de la creación del nuevo marco de pila. Se demuestra así que la pila crece hacia posiciones de memoria descendentes.

11.2. ALINEAMIENTO DE DATOS EN LA PILA

Este ejercicio consistirá en demostrar que las reglas de alineamiento también aplican a los datos contenidos en la pila. Para ello, escribiremos el código necesario para demostrar que cada tipo básico está alineado a posiciones de memoria que son múltiplos de su tamaño. En este caso no utilizaremos KDbg, sino que emplearemos el propio programa para imprimir la dirección de memoria en la que cada una de las variables está almacenada. Así, el siguiente código se ha escrito con el propósito de determinar si los datos están alineados a posiciones de memoria múltiplos de su tamaño y, en ese caso, en qué posiciones.

```
void setup() {
Serial.begin(9600);
}
void loop() {
int i=1;
char a='a';
int j=5;
Serial.println((long) &i, HEX);
Serial.println((long) &a, HEX);
Serial.println((long) &j, HEX);
delay(10000);
}
```

Como se puede observar, se trata de un programa muy sencillo destinado únicamente a imprimir las direcciones de memoria donde están almacenadas las tres variables automáticas i, a y j. Para ello, utilizamos el operador & que nos permite obtener la dirección de memoria del objeto al que se aplica, en este caso, la dirección de memoria de las tres variables.

Si habilitamos el monitor serie tal y como se describía en la sesión 1 podremos observar como el resultado obtenido es el que se muestra en la Figura 35.

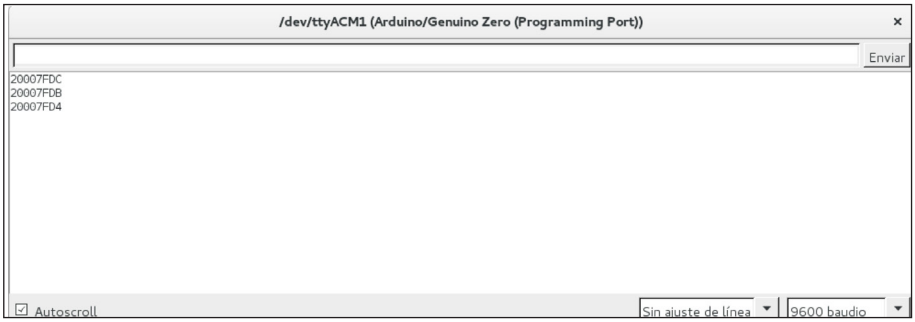

Figura 35. Salida por el puerto serie de la direcciones de la variables automáticas

Para entender mejor cómo se organizan en la pila esas tres variables hemos recurrido a la Figura 36 para representar su localización en memoria y los bytes de relleno que han sido necesarios para satisfacer las reglas de alineamiento.

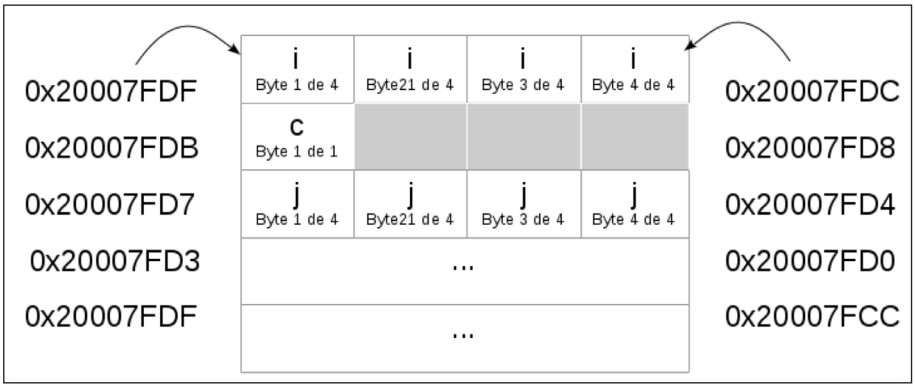

Figura 36 Alineamiento de las tres variables automáticas en memoria

Finalmente, hemos utilizado el depurador KDbg para comprobar cuál es el valor del puntero de pila después de la impresión de las tres variables. Teniendo en cuenta que el puntero de pila siempre apunta a la siguiente dirección de memoria sobre la que se escribirá el dato (predecremento), el valor del SP será 0x20007fd0.

12
SESIÓN 4: EVALUACIÓN

12.1. SESIÓN 4: PREGUNTAS

- **Pregunta 1**. Determina la configuración menos óptima de una estructura que se componga de dos variables tipo double, dos tipo char y una tipo int.
- **Pregunta 2.** ¿Cuánto ocupa en memoria?
- **Pregunta 3.** ¿Cómo la optimizarías para reducir su espacio?
- **Pregunta 4.** ¿Podrías escribir un programa que ocasionara un *stack overflow*?
- **Pregunta 5.** ¿Cuántas iteraciones necesitaría antes de dejar de funcionar?

12.2. SESIÓN 4: RESPUESTAS

12.2.1. PREGUNTA 1

Conociendo las reglas de alineamiento que establece el ABI al respecto del alineamiento de los tipos básicos y compuestos, la configuración menos óptima sería aquella que necesitara más bytes de relleno. Así, teniendo en cuenta que los tipos básicos tienen que estar alineados a direcciones de memoria múltiplos de su tamaño, la configuración menos óptima sería la siguiente:

```
struct ejercicio1 {
char c1;
double d1;
char c2;
double d2;
int i;
};
```

Para comprobar que el alineamiento se lleva a cabo, hemos escrito el siguiente programa que nos permite conocer la dirección de memoria en la cual encontramos a cada uno de los elementos que componen la estructura:

```
void setup() {
Serial.begin(9600);
}

struct ejercicio1 {
char c1;
double d1;
char c2;
double d2;
int i;
};

void loop() {
struct ejercicio1 ej;
Serial.println((long) &ej.c1, HEX);
Serial.println((long) &ej.d1, HEX);
Serial.println((long) &ej.c2, HEX);
Serial.println((long) &ej.d2, HEX);
Serial.println((long) &ej.i, HEX);
delay(10000);
}
```

La salida por el puerto serie muestra las siguientes direcciones:

0x20007fb8
0x20007fc0
0x20007fc8
0x20007fd0
02x0007fd8

12.2.2. PREGUNTA 2

Si representamos gráficamente los elementos de la estructura con la posición que los mismos ocupan dentro de la pila podremos observar, como se ve en la Figura 37 los bytes de relleno necesarios para alinear los tipos de datos básicos que componen la estructura.

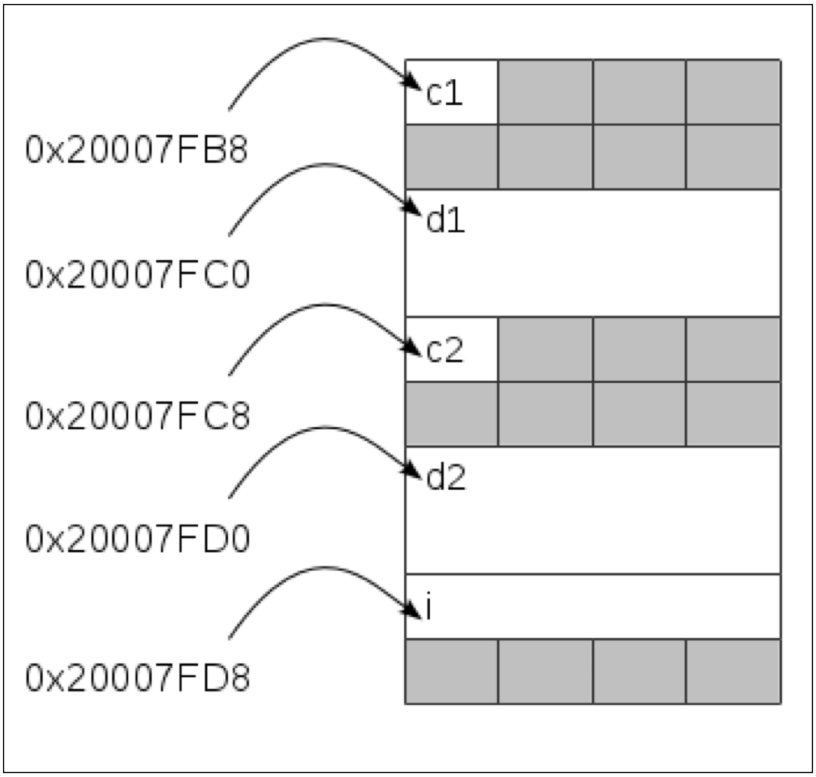

Figura 37. Representación del alineamiento menos óptimo para la estructura propuesta

La estructura ocupa un tamaño de 40 bytes, de los cuales, 18 son bytes de relleno.

12.2.3. PREGUNTA 3

Si reescribimos el código para optimizar ahora su tamaño deberíamos obtener una estructura que ocupe la suma del tamaño de todos los tipos que la componen más los bytes necesarios para redondearla a un número múltiplo del tipo de mayor tamaño. En este caso, la suma de todos los tipos básicos es 2 bytes (uno por cada tipo char), más 16 bytes (8 por cada tipo double) más 4 bytes del tipo int. La suma asciende a 22 bytes pero, siendo que el tipo de mayor tamaño es el double de 8 bytes, la estructura se habrá de redondear al múltiplo de 8 más cercano, en este caso 24.

La Figura 38 representa ahora la configuración (al menos, una de ellas) más óptima para una estructura que contenga los tipos básicos indicados.

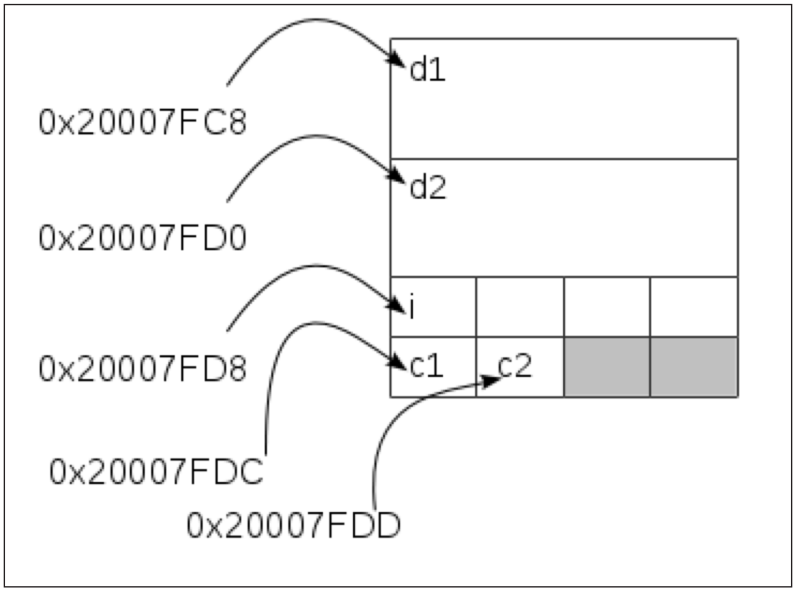

Figura 38. Representación del alineamiento más óptimo para la estructura propuesta

12.2.4. PREGUNTA 4

Como se describe en [tag15], en el caso más simple, un *stack overflow* implica que el código de nuestro programa está consumiendo más espacio en pila del que se ha reservado para la misma[16]. La respuesta del programa en estos casos no es predecible, pudiendo corromper los datos del espacio de memoria reservado para el montículo, pero pudiendo también alcanzar el espacio de las variables globales o estáticas, llegando incluso al colapso del programa en ejecución.

Las llamadas recursivas no controladas son el foco más común de fallos tipo *stack overflow*, porque la pila reserva memoria para el *stack frame* en cada iteración y nunca se libera porque nunca se alcanza el fin de la recursión.

En nuestro caso hemos escrito un programa de prueba que nos servirá para calcular cuántas iteraciones, en el mejor de los casos, soportaría el programa antes de colapsarse.

16 En el caso del Cortex M0+, sabemos que el espacio reservado para las variables globales, la pila y el montículo es de 32KB, muy por encima de la media del resto de Arduinos.

```
void setup() {
Serial.begin(9600);
}

void loop() {
int i=0;
i = incrementa(i);
}
int incrementa(int a){
a = a+1;
Serial.print("Iteracion:");
Serial.println(a);
delay(10000);
incrementa(a);
}
```

Como podemos observar el programa es muy sencillo pero incurre en una recursividad infinita, por cuanto nunca se sale de la función *incrementa*.

12.2.5. PREGUNTA 5

Utilizaremos ahora el depurador KDbg para comprobar cuánto espacio se reserva en la pila por cada llamada. En este caso no será necesario ejecutar el programa a depurar pues sólo nos interesa el código ensamblador asociado a la llamada a la función *incrementa*() para comprobar el tamaño reservado. Así, la Figura 39 muestra las instrucciones asociadas a la llamada a la función.

```
-   12 int incrementa(int a){
       0x2148 push     {r7, lr}
       0x214a sub      sp, #8
       0x214c add      r7, sp, #0
       0x214e str      r0, [r7, #4]
+   13   a = a+1;
```

Figura 39. Ensamblador asociado a la llamada a la función incrementa

Como se puede observar, la llamada a la instrucción push inserta dos elementos en la pila: el registro r7 y el lr. Seguidamente desplaza el puntero a pila en 8 posiciones, restando 8 al valor actual de la pila[17]:

17 Recuérdese que la pila crece hacia posiciones de memoria descendentes y que el Cortex M0+ implementa un modelo con predecremento.

```
sub sp, #8
```

Por lo tanto, cada llamada a la función consumirá 8 bytes. En el mejor de los casos, sólo unos cuantos bytes estarán siendo ocupados para variables automáticas, como en el caso del objeto *Serial,* así que, sin ser excesivamente rigurosos y obviando esos bytes, podemos considerar que si la SRAM tiene un total de 32KB y cada llamada consume 8 bytes, podremos realizar unas 4096 iteraciones antes de que el programa deje de funcionar.

13
SESIÓN 5: LENGUAJE MÁQUINA

El desarrollo de un programa completamente en código ensamblador tiene una complejidad asociada que escapa al propósito de esta asignatura. Sin embargo, tener unas nociones básicas de los fundamentos de programación en ensamblador resulta realmente útil para tareas de optimización de código y depuración. Esta sesión estará dedicada a la familiarización de la programación en ensamblador viendo cómo realizar las tareas más básicas de programación en este lenguaje, como operaciones de control de flujo de programa, llamada a procedimientos o el acceso a datos. Veremos también la posibilidad que el lenguaje C para Arduino nos ofrece a la hora de insertar código en ensamblador directamente en el código de nuestro programa. Esta posibilidad es realmente útil cuando se quiere optimizar un determinado aspecto de nuestro de programa y el compilador no lo hace apropiadamente o las funciones disponibles son más complejas y pesadas de lo que realmente se necesita en ese momento.

13.1. CONTROL DEL PROGRAMA

Una de las operaciones más básicas que podemos utilizar en nuestro programa cuando queremos controlar la ejecución del mismo en base a una determinada condición es el uso de la sentencia *if,* como en el ejemplo que se muestra a continuación.

```
if (counter > 10) then
counter = 0
else
counter = counter + 1
```

Suponiendo que el registro Ro se utilice como la variable counter, el siguiente fragmento en ensamblador se correspondería con el código anterior:

```
CMP R0, #10 ;compare to 10
 BLE incr_counter ;if less or equal, then branch
 MOVS RO,#0 ;counter =0
 B counter_done ; branch to counter_done
incr_counter ADDS R0, R0, #1 ;counter = counter +1
counter_done
```

Otra operación bastante común en un programa es la iteración, con bucles tipo *for*, como por ejemplo:

```
Total = 0;
for (i=0;i<5;i=i+1)
Total = Total + i;
```

Suponiendo ahora que la variable Total está en el registro Ro y la variable i en el registro R1 el bucle del fragmento anterior se podría traducir al siguiente fragmento en ensamblador:

```
MOVS R0,#0 ;Total=0
 MOVS R1, #0 ;i=0
loop ADDS R0,R0,R1 ;Total=Total+i
 ADDS R1, R1, #1 ; i=i+1
 CMP R1, #5 ;compara i con 5
 BLT loop ;Si comparación es menor, entonces bifucar a loop
```

13.2. INTRODUCIENDO RETRASOS EMPOTRANDO ENSAMBLADOR

El uso de la temporización es fundamental para controlar la periodicidad con la que se ejecuta una determinada tarea. Hemos podido comprobar como en el

código escrito hasta ahora para nuestro Arduino la llamada a la función delay nos permitía dejar un margen entre cada iteración de la llamada a la función loop().

El propósito de esta sesión de prácticas no es experimentar con los temporizadores o *timers*, sino familiarización con el lenguaje máquina. Sin embargo, utilizaremos la temporización como excusa o marco justificativo para explicar cómo empotrar código en ensamblador dentro de nuestro programa escrito en el lenguaje basado en *Wiring* de Arduino, utilizando el IDE de Arduino.

El retraso o *delay* más pequeño que podemos introducir en nuestro programa utilizando el lenguaje de Arduino es de 2 µs[18], utilizando para ello la llamada a la función delayMicroseconds().

¿Qué ocurre si se desea producir una espera en un programa inferior a esos 2 o 3 µs? ¿No es posible, acaso? Para dar respuesta a esta necesidad no podremos recurrir a ninguna librería sino que tendremos que utilizar código en ensamblador empotrado en nuestro programa.

Una de las instrucciones más sencillas es la de *nop* o *no operation*. Esta instrucción se ejecuta en un ciclo de reloj y no supone ningún cambio en el estado del procesador y, por lo tanto, no afectará en absoluto a la ejecución del programa. Así, con un procesador que trabaje a 16 MHz, una instrucción que consume un ciclo de reloj tardará 62.5 ns en ser ejecutada. Por lo tanto, este será el menor retraso que podremos introducir en nuestro programa.

Veamos entonces cómo integrar el código ensamblador en nuestro programa, para lo cual utilizaremos la llamada a la función asm, como:

```
asm("código ensamblador");
```

o también como:

```
__asm__("código ensamblador");
```

Así, la sentencia de ensamblador *inline* más básica que podemos utilizar es la de no operación, de la siguiente manera:

```
asm("nop \n");
```

18 El manual de referencia de Arduino establece que esta función es muy precisa a partir de los 3 microsegundos en adelante, pero no pueden asegurar lo mismo para retrasos más pequeños.

Si esta sentencia formara parte del ejemplo del *"Hola, Mundo"* que utilizamos en la primera sesión, el resultado sería el siguiente:

```
void setup() {
Serial.begin(9600);

}
void loop() {
Serial.print("Hello, World");
asm("nop \n");
}
```

13.3. MODOS DE DIRECCIONAMIENTO

Como podemos observar, la instrucción *nop* es una instrucción sin operandos. Este no es siempre el caso y como se ha estudiado podemos encontrar instrucciones con un operando, dos, tres o más. Además, hay diferentes formas de especificar cómo obtener el dato o datos que serán empleados como operandos por la instrucción. La forma más básica de acceso, ya sea para lectura o escritura, es la que utiliza un direccionamiento directo, así por ejemplo, la instrucción *nop* utilizada en la sección anterior puede también ser expresada como una instrucción que mueve el registro ro al ro, de la siguiente manera:

```
asm("mov r0,r0");
```

Esta instrucción utiliza un direccionamiento directo a registro porque los datos se encuentran en los registros especificados directamente.

Utilizaremos ahora el ejemplo del *"Hola, Mundo"* para explorar los modos de direccionamiento que se implementa en este programa. Para ello, utilizaremos la herramienta KDbg para explorar el código ensamblador asociado a cada instrucción en C.

La Figura 40 muestra el código ensamblador asociado a cada una de las sentencias de nuestro código.

Así, podemos ver como en la llamada a la función setup (línea 17) lo primero que encontramos es una instrucción *push* que como sabemos se utiliza para insertar datos en la pila, en este caso los datos del registro r3 y lr. En esta instrucción encontramos un direccionamiento a pila, donde uno de los operandos (el de destino) es implícito por cuanto no hay que indicar dónde se almacenarán los valores de r3 y lr. Por tratarse de una instrucción *push* la unidad de control

```
17 void setup() {
   0x212c push     {r3, lr}
18   Serial.begin(9600);
   0x212e movs     r1, #150        ; 0x96
   0x2130 ldr      r0, [pc, #8]    ; (0x213c <setup()+16>)
   0x2132 lsls     r1, r1, #6
   0x2134 bl       0x29e4 <Uart::begin(unsigned long)>
19 }
   0x2138 pop      {r3, pc}
   0x213a nop                      ; (mov r8, r8)
   0x213c lsls     r4, r6, #2
   0x213e movs     r0, #0
20
21 void loop() {
   0x2110 push     {r3, lr}
22   Serial.print("Hola, Mundo");
   0x2112 ldr      r1, [pc, #16]   ; (0x2124 <loop()+20>)
   0x2114 ldr      r0, [pc, #16]   ; (0x2128 <loop()+24>)
   0x2116 bl       0x276c <Print::print(char const*)>
23   delay(1000);              // wait for a second
   0x211a movs     r0, #250        ; 0xfa
   0x211c lsls     r0, r0, #2
   0x211e bl       0x2280 <delay>
24 }
25
```

Figura 40. Código ensamblador asociado a las sentencias en C

encargada de ejecutar la instrucción ya sabe que el destino será la cima de la pila. Sabemos además que el modelo de pila implementado por el Arduino Zero es un modelo *full-descendent* o lo que es lo mismo, una pila que crece hacia posiciones de memoria descendentes y cuyo puntero está predecrementado.

En la siguiente instrucción encontramos un operando con direccionamiento directo a registro y otro con direccionamiento inmediato.

```
movs r1, #150
```

El valor inmediato es 150 y lo que esta instrucción hace es asignar al registro r1 el valor 150.

La siguiente instrucción `ldr r0, [pc, #8]` implementa un direccionamiento relativo a registro base, que en este caso es el contador de programa o PC con desplazamiento. Esta instrucción carga en el registro ro el contenido que encontremos en la dirección de memoria almacenada en el registro PC, desplazada en 8 posiciones. Como se puede observar en el comentario asociado a esa instrucción, la dirección de memoria donde encontraremos el dato será la 0x213c.

La última instrucción de ese bloque es una instrucción de bifurcación *bl* que utiliza un operando con direccionamiento directo a memoria pues, como se puede observar, indica la dirección de memoria a la que habrá que saltar (previa actualización del contador de programa) para continuar con la ejecución.

Del bloque de instrucciones en ensamblador correspondiente a la sentencia de la línea 18 destacamos la instrucción `lsls r1, r1, #6`. Se trata de una operación de desplazamiento a nivel de bits que, en este caso, desplaza el contenido del registro r1 6 bits a la izquierda. Como podemos observar, esta instrucción tiene tres operandos, los dos primeros con un direccionamiento directo a registro y el tercero con un direccionamiento inmediato. El resto de instrucciones presentan los modos de direccionamiento ya comentados.

14
SESIÓN 6: EVALUACIÓN

14.1. SESIÓN 6: PREGUNTAS

Respecto del siguiente fragmento de código, responde a las siguientes preguntas:

```
void setup() {
Serial.begin(9600);

}
void loop() {
Serial.print("Hello, World");
delay(1000);
}
```

Pregunta 1. Identifica una instrucción que utilice un modo de direccionamiento relativo al contador de programa.

- **Pregunta 2.** Siguiendo la traza de ejecución del programa, identifica una instrucción en ensamblador que se corresponda con una bifurcación condicional y coméntala.
- **Pregunta 3.** Identifica y analiza el contexto en el que se utiliza una instrucción con un operando que utilice direccionamiento indirecto a registro.
- **Pregunta 4.** Identifica una instrucción que utilice un operando con direccionamiento relativo al *stack pointer*, SP o puntero a pila.

- **Pregunta 5.** Identifica un fragmento de código ensamblador que se corresponda con un bucle y analízalo.

14.2. SESIÓN 6: RESPUESTAS

14.2.1. PREGUNTA 1

Lo primero que tendremos que hacer es copiar el código proporcionado por el enunciado del problema y pegarlo en un nuevo archivo (*sketch*) en el IDE de Arduino para, posteriormente, compilarlo y cargarlo en la placa. Una vez hecho esto, el siguiente paso consistirá en seguir los pasos para la depuración remota descritos en el Anexo B. Recuerda que es importante haber activado la optimización de la generación de código ensamblador al mínimo nivel, nivel cero.

Desde el depurador *KDbg* comenzaremos a seguir la traza de ejecución del programa para lo cual iremos siguiendo paso a paso la ejecución a nivel de instrucción en ensamblador *(Step into by instruction)*. De esta forma, cuando la ejecución salte a funciones que forman parte de librerías externas a nuestro propio código, como ocurre por ejemplo cuando llamamos a Serial.begin(9600) o Serial.print podremos ver el código ensamblador asociado.

Así, para dar respuesta a esta pregunta encontramos en el bloque de instrucciones en ensamblador correspondiente a la sentencia de la línea 22, la siguiente instrucción:

```
0x2130 ldr r2, [pc, #24] ; (0x214c <loop()+32>)
```

Se trata una instrucción de carga a un registro, más concretamente, esta instrucción carga un valor en el registro r2. El segundo operando de esta instrucción se trata de un operando relativo al contador de programa, con un desplazamiento de 16 bits. Utilizando la funcionalidad que nos ofrece KDbg para explorar la memoria, podemos comprobar con antelación el valor que se cargará en r2.

En primer lugar, el valor del contador de programa o registro PC es la misma dirección en la que se encuentra la ejecución del programa, más concretamente en la dirección 0x2130. Habrá que aplicar un desplazamiento de 24 bits, lo que da como resultado la dirección de memoria 0x2148. Como se puede observar, si intentamos acceder a esa posición de memoria obtenemos un contenido no alineado al límite de la palabra de 32 bits, sino de 16. Esta posición debe por lo tanto redondearse al límite de la palabra de 4 bytes, dando como resultado la posición 0x214c. Ahora si exploramos esa dirección podemos comprobar como

el contenido de esa posición de memoria se corresponde con el valor cargado en el registro r2 y, más concretamente, con el valor 0x200000b4.

Este bloque de instrucciones está preparando la llamada a la función y lo que estas instrucciones de carga están realizando es preparando el paso de argumentos.

14.2.2. PREGUNTA 2

Para responder a esta pregunta seguiremos con la traza de la ejecución de este programa, hasta la llamada a la función Print::write cuya declaración podemos encontrar en el archivo Print.h. Podemos seguir la traza de llamadas a función en la ventana de la Pila.

Como podemos observar en la definición de la función write, en el bloque de instrucciones ensamblador asociado a la instrucción `if (str == NULL) return 0;` la segunda instrucción es una instrucción de salto condicional:

```
0x276c beq.n 0x277e <Print::write(char const*)+26>
```

Esta instrucción testea si el bit de signo N del registro de estado APSR está activado, lo que indica que la operación anterior de resta, r0 es menor que r1:

```
0x276a subs r0, r1, #0
```

En ese caso, si el bit N está activado se salta a la dirección de memoria 0x277e.

14.2.3. PREGUNTA 3

Dentro de la misma función de la pregunta anterior, podemos ver como en el fragmento de código ensamblador asociado a la sentencia de la línea 50 del fichero Print.h, la segunda instrucción es una de carga de registro que utiliza para el segundo operando un direccionamiento indirecto a registro.

```
0x2772 ldr r3, [r5, #0]
```

Esta instrucción carga en el registro r3 el contenido de la dirección de memoria almacenada en el registro r5, sin aplicar ningún desplazamiento.

El registro r5 contiene el valor 0x200000b4. Si accedemos a esa dirección de memoria utilizando el KDbg vemos que lo que tiene es la dirección 0x4060 <vtable for Uart+8>. La explicación a esta instrucción se entenderá mucho mejor después de haber estudiado el capítulo sobre entrada/salida, aunque en este momento

bastará con adelantar que el mecanismo de entrada/salida implementado por el Arduino Zero es el de registros mapeados en memoria, lo que quiere decir que en nuestro caso, el uso de un periférico como el puerto serie o UART se hará a través de la lectura y escritura en una serie de registros[19] que estarán mapeados en memoria. Esto se traducirá en lecturas y escrituras de posiciones de memoria.

14.2.4. PREGUNTA 4

Continuando con la traza de la ejecución del programa llegamos a la función Print::write definida en el archivo Print.cpp. En su línea 33 podemos observar una sentencia iterativa, tipo *while* con un fragmento de código ensamblador asociado cuya segunda instrucción es la siguiente:

```
0x2748 ldr r3, [sp, #4]
```

Como se puede observar, se trata de una instrucción que emplea un modo de direccionamiento relativo al puntero a pila con un desplazamiento de 4 bits para su segundo operando.

Aunque, como hemos estudiado, la pila es una estructura tipo LIFO (*last in first out*, es posible acceder a valores que están localizados en otras posiciones de la pila utilizando este mecanismo de direccionamiento relativo al puntero a pila. De esta manera podemos acceder al elemento situado justo en la cima de la pila (ya que recordemos que el puntero a pila tenía un valor predecrementado) sin necesidad de sacar el elemento de la pila, o lo que es lo mismo, sin modificar el puntero a pila.

14.2.5. PREGUNTA 5

Para dar respuesta a esta pregunta podemos utilizar el mismo fragmento de código ensamblador analizado en la pregunta anterior. Asociado a la sentencia while (*size–*) de la línea 50 tenemos el siguiente fragmento de código ensamblador.

```
0x2746 adds r4, r1, #0
0x2748 ldr r3, [sp, #4]
0x274a subs r7, r4, r5
0x274c cmp r4, r3
0x274e beq.n 0x2760 <Print::write(unsigned char const*, unsig-
ned int)+36>
```

19 Es importante tener muy presente que estos registros no tienen nada que ver con los registros del procesador

La primera instrucción aunque es una instrucción de suma, en este caso está siendo utilizada como una instrucción para cargar un valor en un registro, más concretamente para cargar en el registro r4 el valor de r1. Si analizamos lo que hay en r1 podemos comprobar como es una dirección de memoria, la 0x3c2c, a partir de la cual está almacenada la cadena *"Hola, Mundo"* como podemos comprobar si exploramos la memoria a partir de esa dirección dándole el formato de Byte, tal y como se ve en la Figura 41.

Por otro lado, la siguiente instrucción recupera de la pila (sin llegar a sacar el elemento de la misma) el elemento almacenado en la posición de memoria 0x20007fac que es el resultado de sumar 4 al valor almacenado en el registro SP. El valor que finalmente se cargará en el registro r3 será la dirección de memoria 0x3c37 que, si exploramos la memoria una vez más, podremos comprobar como es el final de la cadena *"Hola, Mundo"*.

La instrucción de comparación `0x274c cmp r4, r3` lo que está haciendo es comparar el valor del principio de cadena con el de final. Cuando el de principio sobrepase al del final ya se habrán impreso todos los caracteres que componen la cadena o, dicho de otra forma, el tamaño de la cadena a imprimir (variable size) será cero y ya se podrá salir de la función write.

Aunque el código de este ejemplo ya había sido analizado en ocasiones anteriores, ahora hemos podido comprobar como la impresión de la cadena se realiza imprimiendo uno a uno los caracteres que conforman la cadena y que están almacenados en memoria en posiciones contiguas.

Memoria

0x3c2c

Dirección								
0x3c2c <_fini+12>	72 'H'	111 'o'	108 'l'	97 'a'	44 ','	32 ' '	77 'M'	117 'u'
0x3c34 <_fini+20>	110 'n'	100 'd'	111 'o'	0 '\000'	0 '\000'	0 '\000'	0 '\000'	0 '\000'
0x3c3c <g_APinDescription+4>	11 '\v'	0 '\000'	0 '\000'	0 '\000'	2 '\002'	0 '\000'	0 '\000'	0 '\000'
0x3c44 <g_APinDescription+12>	4 '\004'	0 '\000'	0 '\000'	0 '\000'	-1 '\377'	0 '\000'	-1 '\377'	-1 '\377'
0x3c4c <g_APinDescription+20>	-1 '\377'	-1 '\377'	11 '\v'	0 '\000'	0 '\000'	0 '\000'	0 '\000'	0 '\000'
0x3c54 <g_APinDescription+28>	10 '\n'	0 '\000'	0 '\000'	0 '\000'	2 '\002'	0 '\000'	0 '\000'	0 '\000'
0x3c5c <g_APinDescription+36>	4 '\004'	0 '\000'	0 '\000'	0 '\000'	-1 '\377'	0 '\000'	-1 '\377'	-1 '\377'
0x3c64 <g_APinDescription+44>	-1 '\377'	-1 '\377'	10 '\n'	0 '\000'	0 '\000'	0 '\000'	0 '\000'	0 '\000'
0x3c6c <g_APinDescription+52>	14 '\016'	0 '\000'	0 '\000'	0 '\000'	8 '\b'	0 '\000'	0 '\000'	0 '\000'
0x3c74 <g_APinDescription+60>	4 '\004'	0 '\000'	0 '\000'	0 '\000'	-1 '\377'	0 '\000'	-1 '\377'	-1 '\377'
0x3c7c <g_APinDescription+68>	-1 '\377'	-1 '\377'	14 '\016'	0 '\000'	0 '\000'	0 '\000'	0 '\000'	0 '\000'

Figura 41. Contenido de la memoria a partir de la dirección contenida en el registro r1

15
SESIÓN 7: ENTRADA/SALIDA

Durante esta sesión de prácticas se llevará a cabo el proceso de integración de diferentes tipos de periférico de entrada y salida en un robot móvil como el empleado en las prácticas. Se seguirá una metodología adaptada de integración continua para asegurar que no se producen fallos durante las fases de integración.

Como periféricos y dentro del ámbito de la robótica educacional se pueden emplear una gran variedad de dispositivos que podemos dividir de forma genérica entre sensores y actuadores. La plataforma de robotica móvil de BQ *Printbot Evolution* incorpora una serie de dispositivos, también disponibles individualmente o en kits de robótica que facilitan su uso y conexionado con plataformas basadas en Arduino. A continuación, se describen los componentes que se emplearán en las prácticas, detallando tanto sus características como ejemplos de prueba para comprobar su correcto funcionamiento y aprender a interaccionar con ellos.

Adicionalmente, es posible conectar más sensores y actuadores a través de la placa *Sensor Shield v5*, que incorpora el robot, o mediante la conexión de nuevos *shields*.

15.1. ENTRADA: SENSORES

El tipo de sensores empleados en una aplicación de robótica móvil suelen tener la función de facilitar la navegación en los entornos en los que se mueve el robot, ya sea detectando impactos contra objetos, midiendo distancias a obstáculos o detectando referencias como por ejemplo líneas.

15.1.1. SENSOR DE ULTRASONIDOS

Este tipo de sensor consta de un emisor que emite una señal de alta frecuencia y un receptor que recibe la señal cuando esta rebota en un objeto cercano. El sensor de ultrasonidos dispone de una entrada (TRI) para indicarle cuando emitir la onda de alta frecuencia y proporciona una señal de retorno (ECH) que consiste en un pulso con una duración proporcional a la distancia a la que está el objeto donde ha rebotado la onda. En algunos modelos ambas señales están asociadas a un único pin y es necesario cambiar su configuración como entrada o salida durante la ejecución del código al realizar medidas.

Se empleará el modelo sensor de ultrasonidos BAT de BQ, cuyas características se listan en laTabla 8.

Tabla 8. Características del sensor BAT de BQ

Característica	**Valor**
Tipo de salida	Digital
Alimentación	5V
Rango de medida	2-500 cm
Resolución	0.2 cm
Ángulo eficaz	15º
Peso	8 gr

Para verificar el funcionamiento del sensor de ultrasonidos se propone el siguiente programa en el que se manda un pulso de 10 µs por el pin 8 (señal TRI) y se ejecuta la función pulseIn() en el pin 9 (ECH) para medir la duración del pulso de retorno. Como la velocidad del sonido es conocida es posible calcular la distancia recorrida por la onda de ultrasonido a partir del tiempo de ida y vuelta. En el programa propuesto la distancia medida se envía por la comunicación serie para que pueda visualizarse en el PC mediante el monitor serie o con el serial plotter existentes en el menú herramientas del IDE de Arduino, siendo necesaria la conexión vía USB con la placa de Arduino.

```
#define TRI_Pin 8 // Definimos el pin donde se conecta la señal TRI
#define ECH_Pin 9 // Definimos el pin donde se conecta la señal ECH

void setup() {
Serial.begin (9600); // Inicializamos la comunicación serie a 9600 ←-
baudios
pinMode(TRI_Pin, OUTPUT); // Definimos el pin 8 como salida (TRI)
pinMode(ECH_Pin, INPUT); // Definimos el pin 9 como entrada (ECH)
}
void loop() {
long duracion, distancia;
digitalWrite(TRI_Pin, LOW);
delayMicroseconds(10);
digitalWrite(TRI_Pin, HIGH); // Mandamos un pulso positivo de 10 us
delayMicroseconds(10);
digitalWrite(TRI_Pin, LOW);
duracion = pulseIn(ECH_Pin, HIGH); // Medimos la duración del pulso ←-
positivo en la senal ECH
distancia = duracion/58; // Calculamos las distancia en cm ←-
dividiendo la duración del pulso por 58 según documentación del ←-
fabricante
Serial.print(distancia); // Mandamos el valor de la distancia ←-
mediante comunicación serie al PC
Serial.println(" cm");
delay(500); // Realizamos una espera de medio segundo
}
```

15.1.2. SENSOR DE INFRARROJO

El sensor de infrarrojos consta de un LED emisor que emite infrarrojos y un fotodiodo que capta la señal reflejada. Este tipo de sensor permite distinguir entre una superficie blanca o negra según la cantidad de luz reflejada, empleándose comúnmente en aplicaciones de robots siguelíneas.

El modelo que se empleará es el Zum Bloq sensor IR de BQ, cuyas características principales se muestran en la Tabla 9. El modelo de sensor empleado dispone de un potenciómetro para fijar el umbral entre las salidas 0 (negro) y 1 (blanco) que proporciona.

Tabla 9. Características del sensor IR Zum Bloq de BQ

Característica	Valor
Tipo de salida	Digital
Alimentación	5V
Peso	4gr

El modelo de sensor empleado incorpora 2 sensores IR para poder programar una aplicación de siguelíneas. El siguiente programa permite verificar el funcionamiento de estos sensores individualmente conectándolos al pin 3 y ajustar el umbral de detección. La lectura puede visualizarse en el propio módulo sensor ya que incorpora un LED para cada sensor IR que informa de su estado. En el programa propuesto la lectura se envía por la comunicación serie para que pueda visualizarse en el IDE de Arduino mediante el monitor serie o con el serial plotter.

```
const int IR = 3;
int Valor_IR=0;
void setup() {
pinMode(IR,INPUT); // Configuramos el pin 3 donde se conectan los - sensor
IR como INPUT
Serial.begin(9600); // Inicializamos la comunicación serie a 9600 ← baudios
}
void loop() {
Valor_IR = digitalRead(IR); // Leemos la entrada analógicos 0 donde
está conectado el sensor IR
Serial.println(Valor_IR); // Enviamos el valor de lectura a través de las comuni-
cacion serie
delay(10); // retraso entre lecturas de 10 ms
}
```

15.1.3. SENSOR DE LUZ

El sensor de luz se basa en el uso de una LDR, es decir, una resistencia cuyo valor óhmico depende de la luz que incide en ella. Este tipo de sensor proporciona, mediante un circuito acondicionador, una señal analógica proporcional a la luz que incide en él, debiendo conectarse a una entrada analógica del microcontrolador. El valor de lectura que se obtiene está comprendido entre 0 y 1023. Se empleará el modelo Zum Bloq sensor de luz de BQ. Las características principales de este sensor se muestran en la Tabla 10.

Tabla 10. Características del sensor de luz Zum Bloq de BQ

Característica	Valor
Tipo de salida	Analógica
Alimentación	5V
Peso	4gr

Mediante el siguiente programa se puede comprobar el correcto funcionamiento de los dos sensores de luz que incorpora el robot. Para ello se realiza una lectura de la entrada analógica 0 y se muestra el valor a través del monitor serie del IDE de Arduino. La lectura del sensor de luz puede también visualizarse en el *serial plotter*, lo cual permite jugar con el sombreado y la iluminación que incide en el sensor para observar las variaciones.

```
int Valor_luz;
void setup() {
Serial.begin(9600); // Inicializamos la comunicación serie a 9600 baudios
}
void loop() {
Valor_luz = analogRead(A0); // Leemos la entrada analógica 0 donde está
conectado el sensor de luz
Serial.println(Valor_luz); // Enviamos el valor de lectura a través de las comuni-
cación serie
delay(10); // retraso entre lecturas de 10 ms
}
```

15.2. SALIDA: ACTUADORES

15.2.1. ZUMBADOR

El zumbador es un tipo de actuador que permite generar sonidos mediante el uso de una salida PWM con la que se puede fijar la frecuencia del sonido a generar. Se empleará el modelo Zum Bloq zumbador de BQ, cuyas características se muestran en la Tabla 11.

Tabla 11. Características del zumbador Zum Bloq de BQ

Característica	Valor
Tipo de salida	Digital (PWM)
Alimentación	5V
Peso	5gr

Para verificar el correcto funcionamiento del zumbador e ilustrar su uso se puede ejecutar el siguiente programa en el que se emplea la función tone(). Esta función permite generar una señal cuadrada de una frecuencia determinada mediante el segundo argumento de la función. En este ejemplo el zumbador hace sonar la escala musical de forma cíclica con una pequeña pausa gracias al empleo de la función noTone() al final de la escala.

```
#define DO 523 // Definimos la frecuencia de cada nota musical
#define RE 587
#define MI 659
#define FA 698
#define SOL 784
#define LA 880
#define SI 988
#define DOs 1047
int Buzzpin = 8; // Definimos el pin al que conectamos el zumbador

void setup() {
}

void loop() {
tone(Buzzpin,DO); // Mediante la función tone() indicamos el pin de salida y la
frecuencia de la señal cuadrada
delay(300); // Retraso de 300 ms para fija la duración de la nota
tone(Buzzpin,RE); // Repetimos para cada nota creando la escala musical
delay(300);
tone(Buzzpin,MI);
delay(300);
tone(Buzzpin,FA);
delay(300);
tone(Buzzpin,SOL);
delay(300);
tone(Buzzpin,LA);
delay(300);
tone(Buzzpin,SI);
delay(300);
tone(Buzzpin,DOs);
delay(300);
noTone(Buzzpin); // Detenemos la generación de la señal cuadrada
delay(300); // durante 300 ms
}
```

15.2.2. SERVOMOTOR

Para mover el robot se emplean dos servomotores de rotación continua que permiten regular la velocidad y sentido de giro de cada rueda del mismo de forma independiente. Las características del servomotor se muestran en la Tabla 12.

Tabla 12. Características del servomotor de rotación continua

Característica	**Valor**
Modelo	SM-S4303R (SPRINGRC)
Control	Digital (PWM)
Alimentación	4.8-6V
Par	3.3/5.1Kg·cm (4.8V/6V)
Velocidad	43/54 rpm (4.8V/6V)
Peso	44gr
Dimensiones	L 42mm / W 20.5mm / H 39.5mm
Ángulo de rotación	360º
Tipo de conector	Tinkerkit

El siguiente ejemplo de código permite mover dos servos (que en el robot van montados en las ruedas). Para ello se emplea la librería Servo.h y la función write() que permite fijar la velocidad de los servos de rotación continua y el sentido de giro siendo 0 el valor para la velocidad máxima en sentido horario, 90 para detener el servo y 180 para velocidad máxima en sentido antihorario.

Los servos de rotación continua disponen de un tornillo acoplado a un potenciómetro que permite calibrarlos. Para ello se debe girar el tornillo de manera que a velocidad 90, es decir detenido, no gire su eje.

```
#include <Servo.h>

Servo miServo_01; // Creamos objeto Servo (se pueden crear hasta 12)
Servo miServo_02; // Creamos otro objeto Servo

void setup() {
miServo_01.attach(8); // asociamos miservo_01 con el servo conectado en el pin
8
miServo_02.attach(9); // asociamos miservo_02 con el servo conectado en el pin
9
}

void loop() {
miServo_01.write(65); // movemos ambos servos con horario a una velocidad
baja
miServo_02.write(65);
delay(2000); // introducimos un retraso de 2 s
miServo_01.write(90); // Detenemos ambos servos
miServo_02.write(90);
delay(2000); // introducimos un retraso de 2 s
miServo_01.write(115);// movemos ambos servos con antihorario a una veloci-
dad baja
miServo_02.write(115);
delay(2000); // introducimos un retraso de 2 s
miServo_01.write(90); // Detenemos ambos servos
miServo_02.write(90);
delay(2000); // introducimos un retraso de 2 s
}
```

15.2.3. MINISERVO

El robot emplea un miniservo para orientar el sensor de ultrasonidos. Este tipo de servomotor, a diferencia de un servomotor de rotación continua, permite posicionar el eje en una posición entre 0 y 180 grados. Las características del miniservo se muestran en la Tabla 13.

Tabla 13. Características del miniservo

Característica	**Valor**
Modelo	EMAX ES08AII (YIN YAN MODEL LTD)
Control	Digital (PWM)
Alimentación	4.8-6 V

Característica	Valor
Par	1.5/1.8 Kg · cm (4.8 V/6 V)
Velocidad	0.12 s/0.10 s/60 grados (4.8 V/6 V)
Peso	8.5gr
Dimensiones	L 23 mm / W 11.5 mm / H 24 mm
Ángulo de rotación	180 grados
Tipo de conector	Tinkerkit

El siguiente programa permite mover el servo de forma incremental entre sus posiciones 60 y 120 grados. Este procedimiento nos permite poder ajustar el montaje del sensor de ultrasonidos que va acoplado sobre el miniservo. En este tipo de servo la función write() indica la posición en grados a la que queremos que se mueva el servo, siendo pues un control de posición lo que se realiza.

```
#include <Servo.h>

Servo miservo_01; // Creamos objeto Servo (se pueden crear hasta 12)

int pos = 0;

void setup() {
miservo_01.attach(8); // asociamos miservo_01 con el servo conectado en el
pin 8
}

void loop() {
for (pos = 60; pos <= 120; pos += 1) { // movemos el servo controlando su
posición entre 0 y 180 grados con saltos de un grado
miservo_01.write(pos); // mandamos al servo la posición contenida en la vari-
able 'pos' para que vaya a ella
delay(50); // esperamos 50 ms para que el servo llegue a la posición indicada
}
for (pos = 120; pos >= 60; pos -= 1) { // movemos el servo controlando su
posición entre 180 y 0 grados con saltos de un grado
miservo_01.write(pos); // mandamos al servo la posición contenida en la vari-
able 'pos' para que vaya a ella
delay(50); // esperamos 50 ms para que el servo llegue a la posición indicada
}
}
```

15.3. ENTRADA/SALIDA POR INTERRUPCIÓN

En las diversas placas Arduino existe la posibilidad de emplear interrupciones externas, como técnica de E/S, asociadas a los pines de entrada y salida digitales. En el caso concreto del Arduino Zero se pueden emplear todas las entradas digitales excepto la 4.

Las interrupciones capturan eventos en los pines de entrada digitales y hacen una llamada a una rutina de servicio a la interrupción (RSI) que se le haya asignado. Estas rutinas o funciones tienen como limitación que no admiten parámetros de entrada en su llamada, ni retornan ningún valor. Generalmente son funciones que implican una ejecución rápida y en el caso de existir varias solo se puede ejecutar una a la vez, pudiéndose ejecutar otras a continuación según la prioridad establecida. Hay que tener en cuenta que funciones típicamente usadas en la programación de Arduino como delay(), que emplean contadores para realizar temporizaciones no funcionarán correctamente dentro las RSI. El paso de datos entre las RSI y el programa principal debe hacerse a través de variables globales que se deben definir como *volatile* para asegurar su correcta actualización.

La principal función para el manejo de interrupciones con Arduino es attachInterrupt(). Esta función, para el caso del Arduino Zero, emplea la siguiente sintaxis:

```
attachInterrupt(pin, RSI, mode) ;
```

Donde los parámetros de entrada son:

- pin: número del pin de entrada digital y de la interrupción asociada (en el caso del Arduino Zero, y a diferencia de otros modelos, ambos números coinciden).
- RSI: rutina de servicio a la interrupción que se llama cuando se produce la interrupción. Esta función no retorna ningún valor, ni tiene parámetros de entrada.
- mode: establece el modo en que se produce la interrupción asociada a una señal de E/S digital. Los valores válidos que puede tomar este argumento son las constantes recogidas en la Tabla 14.

Tabla 14 Distintos modos de interrupción

Mode	**Descripción**
LOW	Se dispara interrupción cuando el pin está a nivel lógico bajo.
HIGH	Se dispara interrupción cuando el pin está a nivel lógico alto.
RISING	Se dispara interrupción cuando en el pin hay una transición de nivel lógico bajo a alto.
FALLING	Se dispara interrupción cuando en el pin hay una transición de nivel lógico alto a bajo.
CHANGE	Se dispara interrupción cuando en el pin hay una transición de nivel lógico (bajo a alto o alto a bajo).

Para deshabilitar una interrupción se emplea la función detachInterrupt(), usando la sintaxis `detachInterrupt(pin)`, donde pin hace referencia al pin e interrupción asociada.

Existen también las funciones noInterrupts() e interrupts() que permiten deshabilitar y habilitar todas las interrupciones, respectivamente. Se pueden emplear en secciones de código que sean sensibles en cuento a su ejecución temporal, y en las que no queramos que se puedan producir interrupciones.

A continuación, se muestra un ejemplo sencillo en el que se modifica el estado del pin 13 cuando se produce un cambio en el estado del pin 2.

```
const byte led_pin = 13;
const byte interrupcion_pin = 2;
voatile byte estado = LOW;

void setup(){
pinMode(led_pin, OUTPUT);
pinMode(interrupcion_pin, INPUT_PULLUP);
attachInterrupt(digitalPinToInterrupt(interrupcion_pin), cambio_estad, CHANGE);
}

void loop() {
digitalWrite(ledPin, state);
}

void cambio_estado() {
estado = !estado;
}
```

16
SESIÓN 8: EVALUACIÓN

Durante esta sesión, una vez conocidos los sensores y actuadores con los que cuenta el robot, nos centraremos en dotarlo de funcionalidad. Más concretamente, dotaremos al robot de la funcionalidad de seguir una línea de color negro de tal forma que el robot tendrá la posibilidad de adaptar su trayectoria de movimiento en función de hacia dónde derive la línea.

Para la versión más básica sólo haremos uso del sensor infrarrojo para detectar cuando el suelo es blanco o negro y en función de esa medición adaptar el movimiento del robot. Sin embargo, el robot se puede optimizar mucho más utilizado otros sensores o lógica programática para hacer que éste esquive obstáculos, recuerde elecciones pasadas para evitar tomar siempre el mismo camino, emplee interrupciones, etc.

16.1. LECTURA DEL SENSOR INFRARROJO

El robot que hemos desarrollado cuenta con dos sensores infrarrojos IR (derecha e izquierda) que serán los que utilicemos para determinar si el robot está o no encima de la línea negra.

Los sensores IR de nuestro robot estarán conectadas los pines 2 y 3. Así, primero definiremos un alias para dichos pines de tal forma que, para la claridad del código, podamos identificar en cada momento qué sensores hay conectados a dichos pines. Además, como parte de la definición global de variables, asociaremos el alias BLACK al valor 0 para una mayor claridad del código.

```
/********* Variables Globales *********/
const int IR_Right = 2;
const int IR_Left = 3;
const int BLACK=0;
```

Después, será necesario indicar que esos dos pines funcionarán como pines de entrada, para lo cual:

```
void setup()
{
pinMode(IR_Right, INPUT);
pinMode(IR_Left, INPUT);
}
```

Una vez configurados los sensores IR, dentro de la función loop escribiremos el código necesario para obtener los valores medidos por estos sensores, para lo cual utilizaremos la llamada a la función digitalRead aplicada al pin en cuestión:

```
void loop()
{
int colorDetectedBy_IR_Right=digitalRead(IR_Right);
int colorDetectedBy_IR_Left=digitalRead(IR_Left);
}
```

El valor obtenido de la lectura de los sensores IR se compararán con el valor BLACK[20] y en función del resultado de la comparación se actuará sobre los servos de una u otra forma.

16.2. CONTROL DE LOS SERVOS

El movimiento del robot vendrá determinado por la actuación de los servos que permitirá regular la velocidad a la que el robot se mueve, así como el sentido de giro de cada rueda. En primer lugar, tendremos que crear objetos de tipo Servo que serán los que nos permitan utilizar las funciones de dicha clase para configurar su funcionamiento. Asimismo, habrá que indicar los pines a los que dichos servos estarán conectados, en este caso al pin 8 y 9.

20 Recuérdese que la variable BLACK funciona como alias del valor 0.

```
Servo miServo_Right;
Servo miServo_Left;

const int SERVO_RIGHT=8;
const int SERVO_LEFT=9;
```

Dentro de la función setup se llevará a cabo la asociación del servo a cada uno de los pines:

```
void setup(){
miServo_Right.attach(SERVO_RIGHT);
miServo_Left.attach(SERVO_LEFT);
}
```

Analizaremos a continuación el código necesario para hacer que el servo funcione de tres maneras diferentes: parar, avanzando en línea recta, girando a la derecha y girando a la izquierda. Con estos cuatro movimientos conseguiremos que el robot implemente la funcionalidad de un siguelíneas.

La función write de la clase Servo nos permitirá mover el robot hacia la derecha (sentido horario) a su velocidad máxima con el valor 0, con el valor 90 detener el servo y con el valor 180 obtener su velocidad máxima en sentido antihorario (izquierda).

```
// Robot girando a la derecha
miServo_Right.write(90);
miServo_Left.write(180);

// Robot girando a la izquierda
miServo_Right.write(0);
miServo_Left.write(90);

//Robot avanzando
miServo_Left.write(180);
miServo_Right.write(0);

// Robot parado
miServo_Left.write(90);
miServo_Right.write(90);
```

16.3. SIGUELÍNEAS

La función siguelíneas se implementa sobre la base de los valores obtenidos por los sensores IR. Así, cuando el sensor IR derecho no detecta negro y el de la izquierda sí es porque el robot se está saliendo de la línea y habrá que reconducirlo hacia el lado donde aún detecta el color negro, en este caso hacia la izquierda. Para ello, basta con dejar la rueda izquierda parada y mover la derecha. En el caso contrario, cuando aún se detecte negro por la derecha, pero no por la izquierda, el giro habrá de hacerse hacia la derecha, dejando el servo derecho fijo y moviendo el de la izquierda. En el caso en que no se detecte negro por ninguno de los dos sensores IR habrá que dejar el robot parado, parando los dos servos y, finalmente, si ambos sensores detectan negro, los dos servos se moverán.

El código completo quedaría de la siguiente forma:

```
#include <Servo.h>

/********* Variables Globales *********/
const int IR_Right = 2;
const int IR_Left = 3;
const int BLACK=0;

Servo miServo_Right;
Servo miServo_Left;

const int SERVO_RIGHT=8;
const int SERVO_LEFT=9;

/***Function declaration ***/

void setup()
{
pinMode(IR_Right, INPUT);
pinMode(IR_Left, INPUT);
miServo_Right.attach(SERVO_RIGHT);
miServo_Left.attach(SERVO_LEFT);
}

void loop()
{
int colorDetectedBy_IR_Right=digitalRead(IR_Right);
int colorDetectedBy_IR_Left=digitalRead(IR_Left);
if(colorDetectedBy_IR_Right==BLACK){
if(colorDetectedBy_IR_Left==BLACK){ //Hay que avanzar
miServo_Left.write(180);
miServo_Right.write(0);
}else{ //Hay que girar a la derecha
miServo_Right.write(90);
miServo_Left.write(180);
}
}else{
if(colorDetectedBy_IR_Left==BLACK){ //Hay que girar a la izquierda
miServo_Right.write(0);
miServo_Left.write(90);
}else{ //Hay que parar
miServo_Left.write(90);
miServo_Right.write(90);
}
}
}
```

ANEXOS

ANEXO A: ARDUINO Y GNU/LINUX

Este anexo describe el proceso de configuración del entorno de desarrollo para la placa Arduino Zero, siguiendo el proceso descrito en [Ard16].

ELEMENTOS NECESARIOS

Para la programación de una placa Arduino Zero sólo se necesitará la placa y un cable USB estándar con un conector micro USB en uno de sus extremos, similar al que podemos utilizar para conectar nuestros teléfonos móviles al USB de un ordenador.

SOFTWARE DE ARDUINO (IDE)

El proceso de instalación del software se describe aquí para distribuciones GNU/Linux, como por ejemplo Ubuntu.

En primer lugar, habrá que descargar la última versión del software de la página de Arduino[21]. Además de las versiones para Linux (32 y 64 bits) también están disponibles las versiones de Windows.

Una vez descargado el archivo, éste se descomprimirá en el directorio desde donde se desee ejecutar el Software o IDE de Arduino. En este caso, supongamos que la instalación se desea realizar en el *home*.

Dentro del directorio donde hemos descomprimido el IDE encontraremos un script de instalación install.sh que tendremos que ejecutar desde un terminal

21 https://www.arduino.cc/en/Main/Software

(véase Figura 42). El script de instalación lanzará un mensaje cuando la instalación se haya completado correctamente.

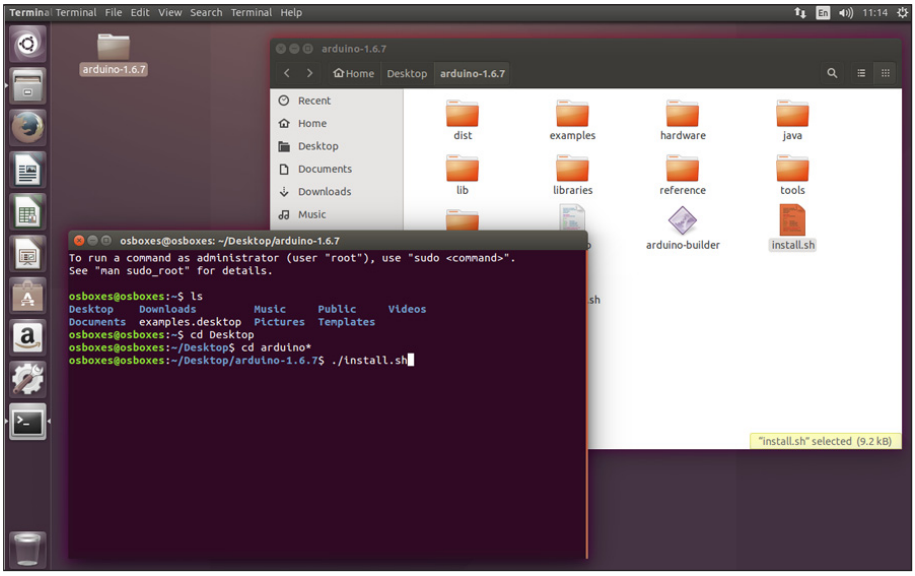

Figura 42. Ejecución del script de instalación

CONFIGURACIÓN

Finalmente, tendremos que dar permisos al usuario que utilizaremos para trabajar con Arduino para que éste pueda tener acceso al puerto de comunicaciones. Para ello debemos ejecutar[22]:

```
$ sudo usermod -a -G dialout <username>
$ sudo chmod a+rw /dev/ttyACM0
```

El IDE de Arduino en la versión empleada durante la redacción de este manual (1.8.1), no trae, entre las tarjetas predefinidas, la del Arduino Zero por lo que habrá que añadirla posteriormente. Para ello, en el menú *Herramientas →Tarjetas→Gestor de Tarjetas* seleccionaremos la correspondiente al Arduino Zero (Arduino SAMD Boards, 32-bits ARM Cortex M0+) y la instalaremos. A continuación es posible ya seleccionar la tarjeta Arduino Zero a través del menú

22 Reemplazar `<username>` con el nombre de usuario. Si se utiliza la máquina virtual de la asignatura, por ejemplo, el nombre de usuario sería alumno

Herramientas→Tarjetas→"Arduino/Genuino Zero (Programming Port)" y también debemos seleccionar el puerto de conexión en *Herramientas→Puerto→"/dev/ttyACMo (Arduino/Genuino Zero (Programming Port))"*. Adicionalmente, para configurar apropiadamente las interfaces software del microcontrolador Cortex Mo y poder usar el puerto de programación (y depuración), ha sido necesario crear una regla para gestionar este dispositivo. Para ello, será necesario seguir los pasos que se detallan a continuación[23].

Debemos crear un fichero en la siguiente ruta *etc/udev/rules.d/98-openocd.rules*, pudiendo utilizar cualquier editor de texto como *gedit*:

```
$ sudo gedit /etc/udev/rules.d/98-openocd.rules
```

Una vez creado, habrá que copiar y pegar el siguiente contenido[24]:

```
ACTION!="add|change", GOTO="openocd_rules_end"
SUBSYSTEM!="usb|tty|hidraw", GOTO="openocd_rules_end" #CMSIS-DAP compatible adapters
ATTRS{product}=="*CMSIS-DAP*", MODE="664", GROUP="plugdev"
LABEL="openocd_rules_end"
```

Con esta regla se da permiso a los usuarios del grupo *plugdev* a que puedan utilizar la placa Arduino. Será necesario comprobar que el usuario forma parte de este grupo, para lo cual podemos usar el comando *groups*:

```
$ groups
```

Si no aparece el grupo *plugdev* se puede añadir con la siguiente línea de comando:

```
$ whoami | sudo xargs usermod -a -G plugdev
```

NUESTRO PRIMER PROGRAMA

El propio IDE de Arduino viene ya con una lista de programas de ejemplo que pueden compilarse y cargarse en la placa. En este caso, siguiendo el manual

23 http://bitofahack.com/post/1437909576

24 Al copiar y pegar el texto de este cuadro, verifica que no se han añadido retornos de carro al final de cada línea.

de la página de Arduino utilizaremos el ejemplo del LED que parpadea como ejemplo de primer programa.

En el menú *Archivo* → *Ejemplos* → 01. *Básicos* → *Blink* seleccionamos el código de ejemplo del led parpadea que nos aparecerá cargado en el IDE. Una vez tengamos el código sólo será necesario compilar y subir el código a nuestra placa, para lo cual bastará con pulsar el botón de *Subir* (véase Figura 43) que se encarga de compilar y subir el código.

Figura 43. Botón para la compilación y carga de un sketch

Si todo ha funcionado correctamente, el LED de la placa comenzará a parpadear una vez cargado el sketch cambiando su estado cada segundo.

ANEXO B: DEPURACIÓN DE PROGRAMAS CON EDBG

Aunque el uso de herramientas de depuración o *debugging* está, generalmente, asociado a la resolución de aquellos problemas que aparecen en tiempo de ejecución, ésta no es su única utilidad. Así, este tipo de herramientas pueden ser utilizadas para explorar la arquitectura en la que un programa está ejecutándose por cuanto el depurador permite detener la ejecución del programa en un determinado punto y explorar el estado del computador (memoria, estado de los registros, valor de las variables, etc.)

La plataforma Arduino Zero presenta, entre sus características más destacables, la incorporación de un depurador integrado en la placa, el EDBG. Hay que tener en cuenta que la depuración en las placas que no cuentan con este módulo requiere de un hardware adicional externo. Llama la atención que, pese a ser uno de los aspectos más novedosos de esta placa, el soporte que Arduino y sus foros de consulta ofrecen para la depuración es nulo. Este anexo ofrece, por lo tanto, información detallada sobre cómo configurar el entorno de desarrollo para poder utilizar el depurador integrado, principalmente como herramienta de exploración de la arquitectura.

En primer lugar, se describirá el proceso necesario para acceder al módulo de depuración, para lo cual se utilizará la herramienta *OpenOCD* (*Open On-Chip-Debugging*)[25]. Esta herramienta es necesaria porque utilizaremos nuestro equipo de trabajo para depurar sobre otra arquitectura diferente, la de Arduino Zero. Por lo tanto, será necesario conectar nuestro equipo al módulo de depuración

25 http://openocd.org/

EDBG de la placa Arduino Zero a través del puerto de programación. *OpenOCD* ofrece un servidor al que nuestro depurador *GDB* podrá conectarse remotamente y a través del cual se enviarán los comandos de control. *OpenOCD* trabajará conjuntamente con el depurador *GDB*, pero teniendo en cuenta que estaremos realizando una depuración cruzada, en lugar de utilizar el depurador GDB nativo para nuestro equipo, tendremos que emplear el de la arquitectura ARM, en este caso el *arm-non-eabi-gdb*. Con estas herramientas, ya sólo será necesario configurar la generación del binario por parte del IDE de Arduino para que pueda ser depurado posteriormente.

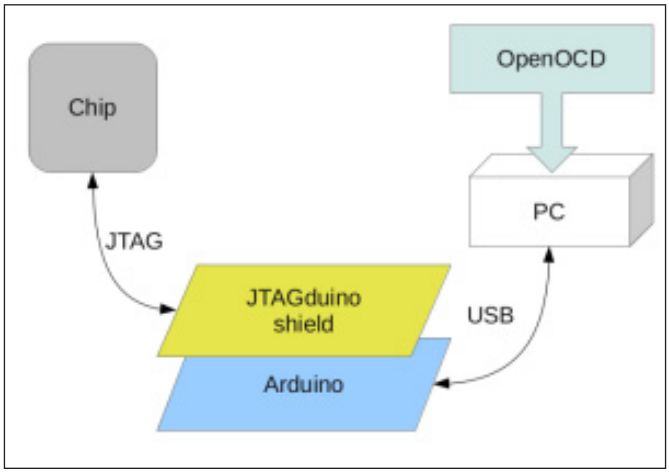

Figura 44. Esquema de conexión de las herramientas de depuración

La Figura 44 describe cómo se interrelacionan entre sí los diferentes elementos implicados en el proceso de depuración sobre Arduino. Como puede observarse, *OpenOCD* corre sobre nuestro ordenador, que será el que esté directamente conectado a la placa Arduino a través del EDBG que será el que ofrecerá la interfaz *JTAG* (*Joint Test Action Group*) para la depuración.

AJUSTES PREVIOS EN EL IDE DE ARDUINO

Aunque parece que desde Arduino están trabajando en la integración del módulo de depuración dentro del IDE, a día de hoy, la depuración debe realizarse desde fuera del mismo. Sin embargo, recurriremos a esta herramienta para el proceso de compilación y generación de binarios, para lo cual será necesario realizar una serie de ajustes tal y como se describe a continuación.

El proceso de compilación genera una serie de archivos que no tienen una localización conocida de antemano. Sería posible revisar los mensajes que se imprimen por la consola, habiendo previamente indicado en las preferencias que se desea "Mostrar salida detallada...". En cualquier caso, nosotros modificaremos el fichero *preferences.txt* para que la salida del proceso de compilación sea a una ruta conocida. Para ello, en el menú *Archivo → Preferencias* editaremos el archivo *preferences.txt* cuya ruta de acceso se muestra en la penúltima línea de la ventana de la ventana de preferencias, tal y como puede observarse en la Figura 45.

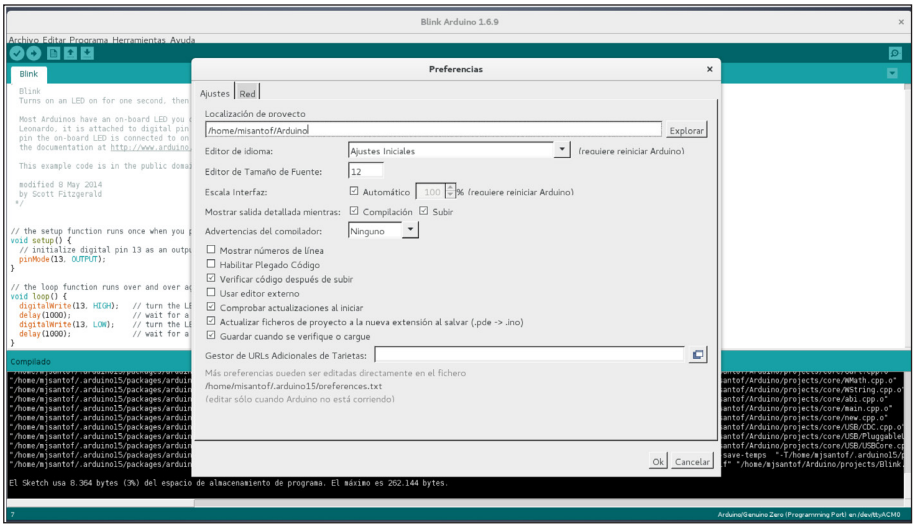

Figura 45. Menú de preferencias del IDE de Arduino

Antes de poder indicar en dicho archivo dónde ubicaremos los ficheros generados como resultado del proceso de compilación, crearemos esa ubicación si previamente no existiera. En nuestro caso, deseamos que todo el código generado se almacene en nuestro home, en el directorio *Arduino/projects*, por lo que bastará con:

```
$ mkdir Arduino/projects
```

Seguidamente podremos añadir la siguiente línea *build.path=Arduino/projects* en el archivo preferences.txt, tal y como puede apreciarse en la Figura 46. Para ello, debemos previamente cerrar el IDE de Arduino, para posteriormente editar el archivo preferences.txt mediante un editor de textos com gedit.

```
$ gedit /home/mjsantof/.arduino15/preferences.txt
```

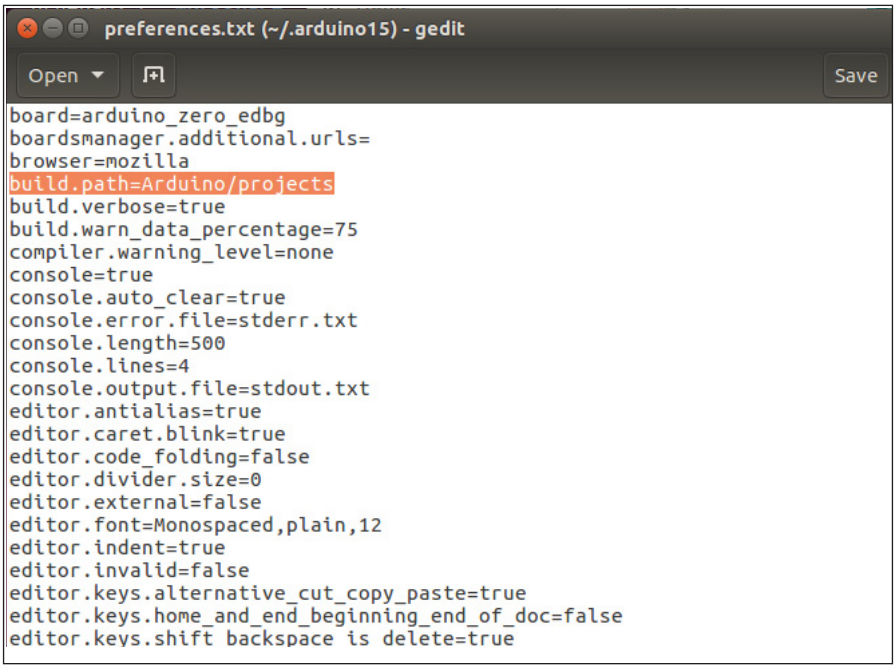

Figura 46. Contenido del archivo preferences.txt y modificación a añadir

Finalmente, para que el proceso de depuración pueda utilizarse para explorar la arquitectura es necesario indicar al compilador que no se desea optimizar el código generado. Por defecto, el IDE de Arduino utiliza un nivel de optimización automático indicado mediante el flag -Os. Nosotros tendremos que eliminar dicha optimización, teniendo en cuenta que este cambio será permanente por lo que si el IDE de Arduino se utilizará para generar otras aplicaciones habría que revertir el cambio para obtener código optimizado.

Para ello editaremos el siguiente fichero:

```
$ gedit /.arduino15/packages/arduino/hardware/samd/1.6.12/platform.txt
```

Buscaremos la línea:

compiler.c.elf.flags=-Os -Wl,–gc-sections -save-temps

y sustituiremos la secuencia -Os por -Oo, quedando por tanto de la siguiente manera:

compiler.c.elf.flags=-Oo -Wl,–gc-sections -save-temps

Después de este último cambio ya podremos generar el código para la depuración, a través del menú *Programa → Verificar/Compilar*. Como resultado, en la ruta indicada, podremos comprobar los ficheros y directorios generados, entre ellos, el de extensión .elf que será el utilizado durante la depuración.

OPENOCD Y GDB

La instalación del *OpenOCD* en sistemas GNU/Linux puede realizarse de la siguiente manera:

```
$ sudo apt-get install openocd
```

Para ejecutar *OpenOCD* es necesario proporcionarle un script con la configuración del hardware, en este caso, del Arduino Zero. *OpenOCD* viene con una serie de scripts listos para usar para otro tipo de plataformas, pero no para Arduino Zero en concreto. No será necesario que nos creemos nuestro propio script porque Arduino ha puesto a nuestra disposición uno, listo para usar, que podremos descargar[26].

```
$ cd /tmp/
$ wget https://github.com/arduino/ArduinoCore-samd/blob/master/variants/arduino_zero/openocd_scripts/arduino_zero.cfg
$ sudo mv arduino_zero.cfg /usr/share/openocd/scripts/target
```

Una vez disponemos del script de configuración pasaremos a conectar el cable USB desde nuestro ordenador a la placa a través del puerto de programación 3. Después podremos lanzar el OpenOCD de la siguiente manera:

```
$ openocd -f /usr/share/openocd/scripts/target/arduino_zero.cfg
```

Si todo ha funcionado correctamente obtendremos la siguiente salida:

26 https://github.com/arduino/ArduinoCore-samd/blob/master/variants/arduino_zero/openocd_scripts/arduino_zero.cfg

```
$ openocd -f /usr/share/openocd/scripts/target/arduino_zero.cfg
Open On-Chip Debugger 0.9.0 (2015-05-28-17:08)
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'swd'
adapter speed: 500 kHz
adapter_nsrst_delay: 100
cortex_m reset_config sysresetreq
Info : CMSIS-DAP: SWD Supported
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : CMSIS-DAP: FW Version = 02.01.0157
Info : SWCLK/TCK = 1 SWDIO/TMS = 1 TDI = 1 TDO = 1 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 500 kHz
Info : SWD IDCODE 0x0bc11477
Info : at91samd21g18.cpu: hardware has 4 breakpoints, 2 watchpoints
```

En este momento el OpenOCD habrá abierto dos puertos, uno para el servidor TELNET en el 4444 y otro para el servidor GDB en el 3333. En nuestro caso depuraremos utilizando la herramienta GDB por lo que tendremos que conectarla al servidor ofrecido por el OpenOCD en el puerto 4444.

El concepto de depuración cruzada se utiliza para referirse al proceso por el cual es posible depurar programas que corren en otra arquitectura diferente de la que está ejecutando el depurador. En nuestro caso utilizaremos un depurador, el GDB, que correrá sobre nuestro ordenador para depurar un programa que estará ejecutándose sobre la placa de Arduino. Por lo tanto, tendremos que utilizar una versión de GDB que sea compatible tanto con la arquitectura sobre la que se ejecutará como sobre la que será objeto de la depuración. La instalación del compilador cruzado para ARM se hará de la siguiente manera:

```
$ sudo apt-get install gdb-arm-none-eabi
```

Una vez instalada la herramienta, veremos el proceso de depuración sobre línea de comandos utilizando únicamente la herramienta GDB. Después, recurriremos

a la herramienta KDbg que es simplemente un *frontend* o interfaz gráfica que simplifica la interacción con el depurador GDB, que corre por debajo.

```
$ arm-none-eabi-gdb
GNU gdb (7.10-1+9) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type “show copying”
and “show warranty” for details.
This GDB was configured as “--host=x86_64-linux-gnu --target=arm-none-
eabi”.
Type “show configuration” for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type “help”.
Type “apropos word” to search for commands related to “word”.
(gdb)
```

El resultado, como se puede ver en el listado anterior, es el intérprete de comandos de GDB. La primera orden a ejecutar será la especificación del código binario que tendrá que ser depurado, para lo cual indicaremos la ruta que anteriormente se había establecido como destino:

```
(gdb) file Arduino/projects/Blink.ino.elf
Reading symbols from Arduino/projects/Blink.ino.elf...done.
(gdb)
```

En este caso, hemos utilizado el código del ejemplo Blink, que viene con el IDE de Arduino. Una vez indicado el binario a depurar solo falta especificar que se trata de una depuración remota en el puerto donde el OpenOCD tiene su servidor GDB. Para ello:

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x00000000 in ?? ()
(gdb)
```

En este momento ya podemos enviar comandos a la placa, pero previamente es necesario tomar el control de la ejecución del programa que está corriendo en la placa. Como puede apreciarse, en la placa sigue corriendo el programa que hace parpadear el LED. Esto indica que el depurador aún no ha tomado el control del programa, ya que para ello deberemos emplear el siguiente comando:

```
(gdb) monitor reset halt
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x81000000 pc: 0x00000858 msp: 0x200023a0
(gdb)
```

En este momento el LED dejará de parpadear. Tendremos que iniciar ahora la ejecución controlada por parte del depurador, para lo cual:

```
(gdb) monitor reset init
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x81000000 pc: 0x00000858 msp: 0x200023a0
(gdb)
```

La opción l lista el código del programa o la función indicada. Por ejemplo:

```
(gdb) l loop
19 // initialize digital pin 13 as an output.
20 pinMode(13, OUTPUT);
21 }
22
23 // the loop function runs over and over again forever
24 void loop() {
25 digitalWrite(13, HIGH); // turn the LED on (HIGH is the voltage level)
26 delay(1000);
// wait for a second
27 digitalWrite(13, LOW); // turn the LED off by making the voltage LOW
28 delay(1000);
// wait for a second
(gdb)
```

"l loop" lista el código de la función loop junto con el número de línea. Este número puede utilizarse para poner un punto de ruptura en la ejecución de la siguiente manera:

```
(gdb) b 25
Breakpoint 1 at 0x2124: file /tmp/Blink/Blink.ino, line 25.
(gdb)
```

Si ahora continuamos con la ejecución del código observaremos como ésta se detiene al llegar a dicha línea. Para continuar la ejecución utilizaremos el comando cont:

```
(gdb) cont
Continuing.
Note: automatically using hardware breakpoints for read-only addresses.
Breakpoint 1, loop () at /tmp/Blink/Blink.ino:25
25 digitalWrite(13, HIGH); // turn the LED on (HIGH is the voltage level)
(gdb)
```

Finalmente, para terminar la depuración bastará con utilizar el comando quit.

KDBG: FRONTEND PARA GDB

El proceso de depuración utilizando la herramienta GDB requiere conocer de antemano los diferentes comandos que se pueden utilizar, así como la manera en la que estos pueden ser utilizados. Para usuarios no experimentados este proceso puede ser demasiado tedioso y lento por lo que el uso de un *frontend* o interfaz gráfica puede resultar de gran utilidad.

Existen varias opciones, pero nosotros hablaremos de la herramienta KDbg. Esta herramienta ofrece una interfaz intuitiva y fácil de utilizar, con diferentes ventanas con las que puede explorarse la plataforma sobre la que está corriendo el programa (registros, memoria, pila, etc.). Ofrece además una serie de botones para controlar la ejecución y la inserción y eliminación de puntos de ruptura. Además, lo que diferencia a esta herramienta de otras es la posibilidad de explorar el código ensamblador en el que cada línea de código es traducida, simplemente desplegando la línea en cuestión, tal y como puede verse en la Figura 47.

La herramienta KDbg debe configurarse para que pueda depurar en remoto un programa que corra en la plataforma Arduino Zero. Para ello, en primer lugar, habrá lanzar la aplicación indicando el puerto al que habrá de conectarse:

```
$ kdbg -r :3333
```

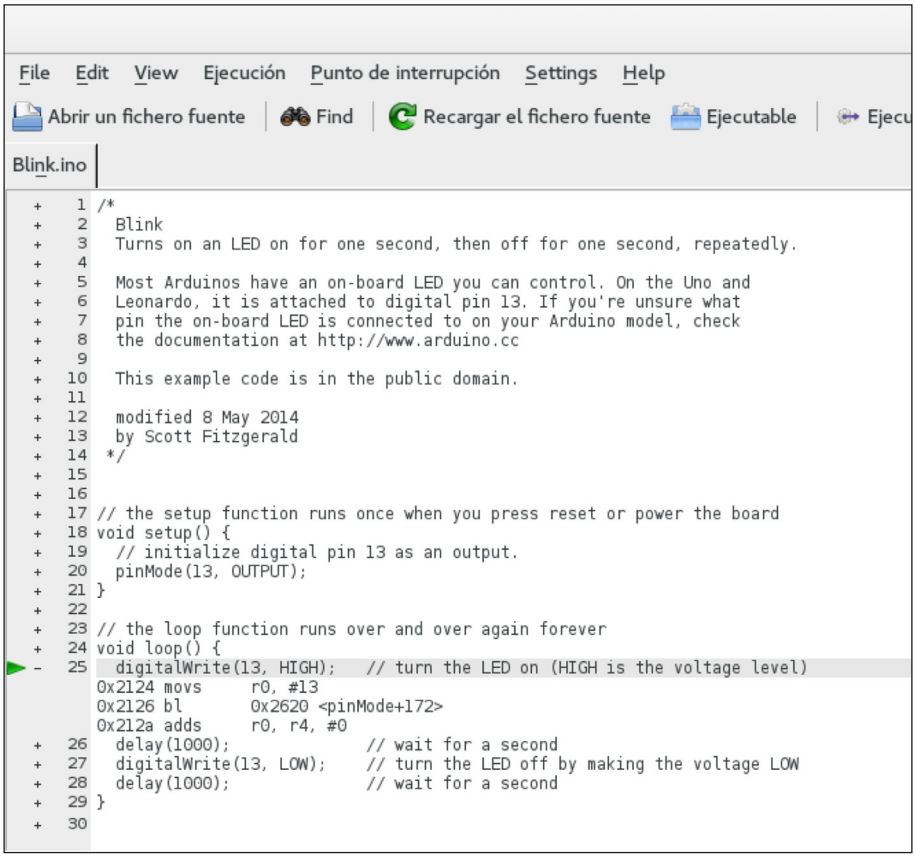

Figura 47. Código ensamblador correspondiente a una línea de código

Es importante que tengamos en cuenta que el servidor GDB del OpenOCD sólo puede atender una conexión a la vez, por lo tanto, habrá que asegurarse de que la ejecución anterior del GDB se ha finalizado.

Lo primero será configurar el KDbg para que utilice el depurador para la arquitectura ARM. Para ello, en el menú *Settings* → *Opciones globales*... en el cuadro de texto que indica cómo iniciar GDB copiaremos la siguiente línea, tal y como se puede observar en la Figura 48:

```
arm-none-eabi-gdb -ex 'monitor reset halt' -ex 'monitor reset
init' -fullname -nx
```

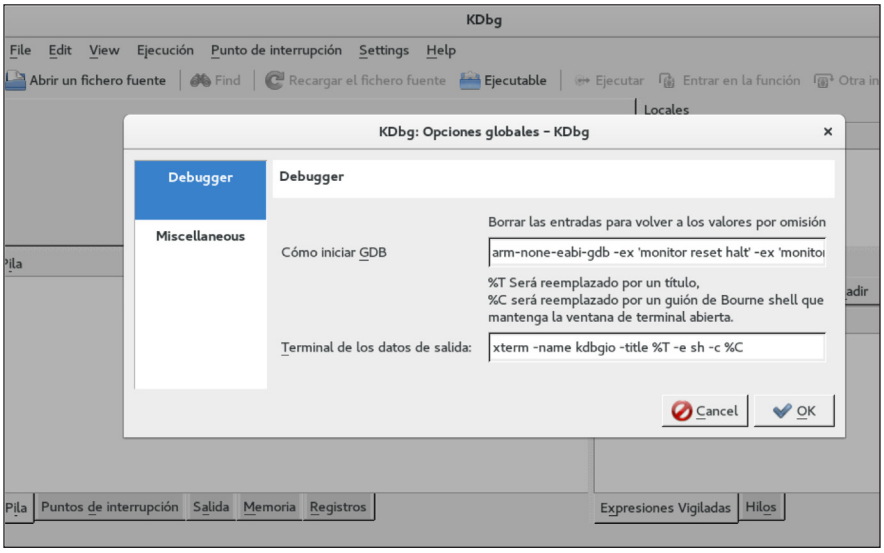

Figura 48. Comando de inicialización del KDbg

Con este cambio estamos indicando que el depurador utilizado es el de la arquitectura ARM y, además, estaremos indicando los comandos necesarios para que se controle la ejecución del programa de la placa, tal y como se hizo al comienzo de la explicación del GDB.

Una vez hechos estos cambios y para que tengan efecto será necesario reiniciar la herramienta, mediante el menú *File* → *Quit*. Después de reiniciar los cambios serán permanentes y por lo tanto, si se desean realizar otras tareas de depuración habrá que restaurar las opciones globales adaptadas para el Arduino Zero.

Después de estos cambios y de haber lanzado el KDbg en remoto tendremos que indicar el ejecutable que será depurado, lo que en el GDB hicimos con el comando file. Para ello pulsamos el botón Ejecutable, como se muestra en la Figura 49, y seleccionamos el archivo .elf a depurar, que en este caso será también el del ejemplo de parpadeo de un LED.

Cuando el archivo se abre también vemos el código del Blink.ino en la ventana de código. La inserción de puntos de interrupción es muy cómoda porque basta con pinchar con el botón derecho sobre la línea sobre la que se desea insertar el punto y selección *Poner/Quitar puntos de interrupción*.

Al igual que ocurría cuando depuramos con el GDB, en este punto, la placa sigue parpadeando, lo que indica que el control no ha sido aún tomado por el KDbg. Si ponemos un punto de interrupción en la línea 25:

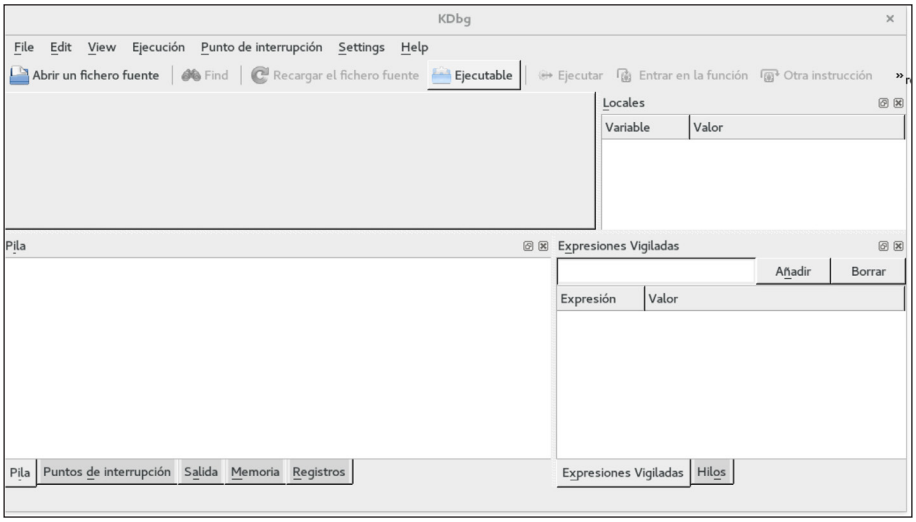

Figura 49. Selección del archivo ejecutable a depurar

```
digitalWrite(13, HIGH); // turn the LED on (HIGH is the voltage level)
```

y a continuación pulsamos Ejecutar, observaremos como el control de la ejecución está ahora en manos del KDbg. La ejecución se detendrá en el punto de ruptura y desde ahí podremos explorar cuestiones como el contenido de los registros o las posiciones de memoria en la que se han cargado las instrucciones, por ejemplo. En este punto, bastará simplemente con utilizar las diferentes opciones: Entrar en la función, Otra instrucción, Salir de la función, etc., para explorar el programa en ejecución.

BIBLIOGRAFÍA

[Ard16] Arduino. *Getting Started with Arduino and Genuino on Linux* [online], 2016. <https://www.arduino.cc/en/Guide/Linux> [Consulta: 15 de agosto de 2016].

[ARM12] ARM. *Cortex-M0+ Technical Reference Manual*, 2012.

[Atm14a] Atmel. *EDBG. User Guide*, 2 2014.

[Atm14b] Atmel. *SAM-BA Bootloader for SAM D21*, 8 2014.

[Whe11] Dale Wheat. *Arduino Internals*. Apress, Berkely, CA, USA, edición 1st, 2011.

[ARM16] ARM. *Application Binary Interface (ABI) for the ARM Architecture* [online], 2016. <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.abi/index.html> [Consulta: 15 de agosto de 2016].

[MFSR10] Francisco Moya Fernandez y Maria J. Santofimia Romero. *LABORATORIO DE ESTRUCTURA DE COMPUTADORES empleando videoconsolas Nintendo DS*. Bubok, 2010.

[ARM15] ARM. *ARM Procedure Call Standard for the ARM Architecture* [online], 2015.<http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf> [Consulta: 15 de agosto de 2016].

[tag15] Front Matter. En Joseph Yiu, editor, *The Definitive Guide to Arm® Cortex®-M0 and Cortex-M0+ Processors (Second Edition)*. Newnes, Oxford, edición Second Edition, 2015.

[Sho16] *Ken Shores. Memory and the Arduino* [online], *2016.* <http://www.sumidacrossing.org/Musings/files/160606_Memory_and_the_Arduino.php> [Consulta: 15 de agosto de 2016].

Este manual de prácticas describe el proceso de desarrollo y programación de un robot móvil empleando Arduino Zero. A lo largo de este proceso el alumnado podrá explorar, de un modo práctico, los aspectos más esenciales de la estructura de un computador (el sistema de memoria, entrada/salida, ABI, etc.).

Ediciones de la Universidad
de Castilla-La Mancha